

4001

8080 PROGRAMMING FOR LOGIC DESIGN



BY ADAM OSBORNE

8080 PROGRAMMING FOR LOGIC DESIGN



Copyright © 1976 by Adam Osborne and Associates, Incorporated

All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publishers.

Published by Adam Osborne and Associates, Incorporated
P.O. Box 2036, Berkeley, California 94702

For ordering and pricing information in Europe contact:

Sole Distributor

SYBEX

313 Rue Lecourbe
75015 — PARIS, FRANCE
Telex: 200858 Sybex

TABLE OF CONTENTS

CHAPTER		PAGE
1	INTRODUCTION	1-1
	WHAT THIS BOOK ASSUMES YOU KNOW	1-2
	UNDERSTANDING ASSEMBLY LANGUAGE	1-2
	HOW THIS BOOK HAS BEEN PRINTED	1-2
2	ASSEMBLY LANGUAGE AND DIGITAL LOGIC	2-1
	THE DESIGN CYCLE	2-1
	SIMULATING DIGITAL LOGIC	2-4
	MICROCOMPUTER SIMULATION OF A SIGNAL INVERTER	2-5
	A MICROCOMPUTER EVENT SEQUENCE	2-5
	IMPLEMENTING THE TRANSFER FUNCTION	2-6
	DETERMINING DATA SOURCES AND DESTINATIONS	2-6
	EVENT TIMING	2-11
	BUFFERS, AMPLIFIERS AND SIGNAL LOADS	2-13
	MICROCOMPUTER SIMULATION OF 7404/05/06/07 HEX INVERTERS	2-19
	MICROCOMPUTER SIMULATION OF 7408/09 QUADRUPLE TWO-INPUT POSITIVE AND GATES	2-20
	TWO INPUT FUNCTIONS	2-21
	THE MICROCOMPUTER SIMULATION OF A 7411 TRIPLE, THREE-INPUT, POSITIVE AND GATE	2-22
	THREE INPUT FUNCTIONS	2-22
	MINIMIZING CPU REGISTER ACCESSES	2-24
	COMPARING MEMORY UTILIZATION AND EXECUTION SPEED	2-26
	THE MICROCOMPUTER SIMULATION OF A 7474 DUAL, D-TYPE, POSITIVE EDGE TRIGGERED FLIP-FLOP WITH-PRESET AND CLEAR	2-24
	A DIGITAL LOGIC DESCRIPTION OF FLIP-FLOPS	2-27
	AN ASSEMBLY LANGUAGE SIMULATION OF FLIP-FLOPS	2-29
	MICROCOMPUTER SIMULATION OF FLIP-FLOPS IN GENERAL	2-30
	THE MICROCOMPUTER SIMULATION OF REAL TIME DEVICES	2-31
	THE 555 MONOSTABLE MULTIVIBRATOR	2-31
	THE 74121 MONOSTABLE MULTIVIBRATOR	2-32
	THE 74107 DUAL J-K MASTER-SLAVE FLIP-FLOP WITH CLEAR	2-34
	MICROCOMPUTER SIMULATION OF REAL TIME	2-35
	MICROCOMPUTER TIMING INSTRUCTION LOOPS	2-35
	THE LIMITS OF DIGITAL LOGIC SIMULATION	2-39
	INTERFACING WITH EXTERNAL ONE-SHOTS	2-39
	TIME OUT AND INTERRUPTS	2-41
3	A DIRECT DIGITAL LOGIC SIMULATION	3-1
	HOW THE QUME PRINTER WORKS	3-2
	INPUT AND OUTPUT SIGNALS	3-8
	INPUT/OUTPUT DEVICES	3-8
	THE 8255 PROGRAMMABLE PERIPHERAL INTERFACE	3-8
	THE 8212 8-BIT INPUT/OUTPUT PORT	3-11
	<u>INPUT SIGNALS</u>	3-12
	RETURN STROBE	3-13
	PFL REL	3-14

TABLE OF CONTENTS (Continued)

CHAPTER		PAGE
	RIB LIFT RDY	3-14
	PW STROBE	3-14
	FFA	3-15
	RESET	3-15
	PFR REL	3-15
	CA REL	3-16
	FFI	3-16
	EOR DET	3-16
	HAMMER ENABLE FF	3-18
	CLK	3-18
	H1 - H6	3-18
	INPUT SIGNAL SUMMARY	3-18
	OUTPUT SIGNALS	3-19
	A DIGITAL LOGIC ORIENTED SIMULATION	3-19
	A LOGIC OVERVIEW	3-20
	FLIP-FLOP FFA _W	3-21
	SIMULATING FLIP-FLOP FFA _W	3-23
	FLIP-FLOP FFB _W	3-29
	SIMULATING FLIP-FLOP FFB	3-31
	FLIP-FLOP FFC	3-35
	SIMULATING FLIP-FLOP FFC	3-37
	START RIBBON MOTION PULSE SIMULATION	3-39
	FLIP-FLOP FFD	3-41
	SIMULATING FLIP-FLOP FFD	3-41
	FLIP-FLOP FFE	3-43
	PW SETTLING ONE-SHOT	3-45
	SIMULATING THE PW ONE-SHOT	3-46
	FLIP-FLOP FFF	3-46
	SIMULATING FLIP-FLOP FFF	3-48
	THE 555 MULTIVIBRATOR	3-51
	SIMULATING MULTIVIBRATOR 555	3-51
	THE PW RELEASE ENABLE FLIP-FLOP	3-58
	SIMULATING THE PW RELEASE ENABLE FLIP-FLOP	3-58
	SIMULATING THE PW READY ENABLE ONE-SHOT	3-60
	SIMULATING SUMMARY	3-63
4	A SIMPLE PROGRAM	4-1
	ASSEMBLY LANGUAGE TIMING VERSUS DIGITAL LOGIC TIMING	4-1
	INPUT AND OUTPUT SIGNALS	4-1
	MICROCOMPUTER DEVICE CONFIGURATION	4-3
	GENERAL DESIGN CONCEPTS	4-3
	8255 PROGRAMMABLE PERIPHERAL INTERFACE (PPI)	4-4
	SYSTEM INITIALIZATION	4-6
	ROM AND RAM MEMORY	4-7
	PROGRAM FLOWCHART	4-9
	PROGRAM LOGIC ERRORS	4-23
	RESET AND INITIALIZATION	4-26

TABLE OF CONTENTS (Continued)

CHAPTER		PAGE
	A PROGRAM SUMMARY	4-27
5	A PROGRAMMER'S PERSPECTIVE	5-1
	SIMPLE PROGRAM EFFICIENCY	5-1
	EFFICIENT TABLE LOOKUPS	5-1
	HARDWARE UTILIZATION	5-5
	HARDWARE-SPECIFIC INSTRUCTIONS	5-5
	DIRECT USE OF HARDWARE FEATURES	5-7
	SUBROUTINES	5-9
	SUBROUTINE CALL	5-11
	SUBROUTINE RETURN	5-15
	WHEN TO USE SUBROUTINES	5-16
	CONDITIONAL SUBROUTINE RETURNS	5-17
	MULTIPLE SUBROUTINE RETURNS	5-19
	CONDITIONAL SUBROUTINE CALLS	5-22
	MACROS	5-22
	WHAT IS A MACRO?	5-23
	MACROS WITH PARAMETERS	5-24
	INTERRUPTS	5-27
	INTERRUPT HARDWARE CONSIDERATIONS	5-27
	INTERRUPT SERVICE PROGRAM	5-29
	JUSTIFYING INTERRUPTS	5-34
	MULTIPLE INTERRUPTS	5-35
6	THE 8080/9080 INSTRUCTION SET	6-1
	ABBREVIATIONS	6-1
	STATUS	6-2
	INSTRUCTION OBJECT CODES	6-2
	INSTRUCTION EXECUTION TIMES AND CODES	6-2
	ACI — ADD WITH CARRY IMMEDIATE TO ACCUMULATOR	6-12
	ADC — ADD REGISTER OR MEMORY WITH CARRY, TO ACCUMULATOR	6-13
	ADD — ADD REGISTER OR MEMORY TO ACCUMULATOR	6-14
	ADI — ADD IMMEDIATE TO ACCUMULATOR	6-16
	ANA — AND REGISTER OR MEMORY WITH ACCUMULATOR	6-17
	ANI — AND IMMEDIATE WITH ACCUMULATOR	6-18
	CALL — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND	6-19
	CC — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND BUT ONLY IF THE CARRY STATUS EQUALS 1	6-20
	CM — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE SIGN STATUS EQUALS 1	6-20
	CMA — COMPLEMENT THE ACCUMULATOR	6-21
	CMC — COMPLEMENT THE CARRY STATUS	6-21
	CMP — COMPARE REGISTER OR MEMORY WITH ACCUMULATOR	6-22
	CNC — CALL THE SUBROUTINE IDENTIFIED BY THE OPERAND, BUT ONLY IF THE CARRY STATUS EQUALS 0	6-24

TABLE OF CONTENTS (Continued)

CHAPTER	PAGE
CNZ — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE ZERO STATUS EQUALS 0	6-24
CP — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE SIGN STATUS EQUALS 0	6-24
CPE — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE PARITY STATUS EQUALS 1	6-25
CPI — COMPARE ACCUMULATOR CONTENTS WITH IMMEDIATE DATA	6-26
CPO — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE PARITY STATUS EQUALS 0	6-26
CZ — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE ZERO STATUS EQUALS 1	6-27
DAA — DECIMAL ADJUST ACCUMULATOR	6-28
DAD — ADD A REGISTER PAIR TO H AND L	6-29
DCR — DECREMENT REGISTER OR MEMORY CONTENTS	6-30
DCX — DECREMENT REGISTER PAIR	6-32
DI — DISABLE INTERRUPTS	6-33
EI — ENABLE INTERRUPTS	6-33
HLT — HALT	6-35
IN — INPUT TO ACCUMULATOR	6-36
INR — INCREMENT REGISTER OR MEMORY CONTENTS	6-36
INX — INCREMENT REGISTER PAIR	6-38
JC — JUMP IF CARRY	6-39
JM — JUMP IF MINUS	6-39
JMP — JUMP TO THE INSTRUCTION IDENTIFIED IN THE OPERAND	6-40
JNC — JUMP IF NO CARRY	6-40
JNZ — JUMP IF NOT ZERO	6-41
JP — JUMP IF PLUS	6-41
JPE — JUMP IF PARITY EVEN	6-41
JPO — JUMP IF PARITY ODD	6-42
JZ — JUMP IF ZERO	6-42
LDA — LOAD ACCUMULATOR FROM MEMORY USING DIRECT ADDRESSING	6-43
LDAX — LOAD ACCUMULATOR FROM MEMORY LOCATION ADDRESSED BY REGISTER PAIR	6-44
LHLD — LOAD H AND L REGISTERS DIRECT	6-45
LXI — LOAD A 16-BIT VALUE, IMMEDIATE, INTO A REGISTER PAIR	6-46
MOV — MOVE DATA	6-46
MVI — LOAD DATA IMMEDIATE INTO REGISTER OR MEMORY	6-48
NOP — NO OPERATION	6-50
ORA — OR REGISTER OR MEMORY WITH ACCUMULATOR	6-51
ORI — OR IMMEDIATE WITH ACCUMULATOR	6-52
OUT — OUTPUT FROM ACCUMULATOR	6-53
PCHL — JUMP TO ADDRESS SPECIFIED BY HL	6-54
POP — READ FROM THE TOP OF THE STACK	6-55

TABLE OF CONTENTS (Continued)

CHAPTER		PAGE
	PUSH — WRITE TO THE TOP OF THE STACK	6-56
	RAL — ROTATE ACCUMULATOR LEFT THROUGH CARRY	6-57
	RAR — ROTATE ACCUMULATOR RIGHT THROUGH CARRY	6-58
	RC — RETURN IF THE CARRY STATUS EQUALS 1	6-59
	RET — RETURN FROM SUBROUTINE	6-59
	RLC — ROTATE ACCUMULATOR LEFT	6-60
	RM — RETURN IF THE SIGN STATUS EQUALS 1	6-60
	RNC — RETURN IF THE CARRY STATUS EQUALS 0	6-61
	RNZ — RETURN IF THE ZERO STATUS EQUALS 0	6-62
	RP — RETURN IF THE SIGN STATUS EQUALS 0	6-62
	RPE — RETURN IF THE PARITY STATUS EQUALS 1	6-63
	RPO — RETURN IF THE PARITY STATUS EQUALS 0	6-64
	RRC — ROTATE ACCUMULATOR RIGHT	6-64
	RST — RESTART	6-65
	RZ — RETURN IF THE ZERO STATUS EQUALS 1	6-65
	SBB — SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR WITH BORROW	6-66
	SBI — SUBTRACT IMMEDIATE DATA FROM ACCUMULATOR WITH BORROW	6-68
	SHLD — STORE H AND L REGISTERS DIRECT	6-69
	SPHL — LOAD THE STACK POINTER FROM THE H AND L REGISTERS	6-69
	STA — STORE ACCUMULATOR IN MEMORY USING DIRECT ADDRESSING	6-70
	STAX — STORE ACCUMULATOR IN THE MEMORY LOCATION ADDRESSED BY A REGISTER PAIR	6-71
	STC — SET CARRY STATUS	6-72
	SUB — SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR	6-72
	SUI — SUBTRACT IMMEDIATE DATA FROM ACCUMULATOR	6-74
	XCHG — EXCHANGE DE AND HL REGISTERS' CONTENTS	6-75
	XRA — EXCLUSIVE-OR REGISTER OR MEMORY WITH ACCUMULATOR	6-76
	XRI — EXCLUSIVE-OR IMMEDIATE DATA WITH ACCUMULATOR	6-77
	XTHL — EXCHANGE TOP OF STACK WITH HL	6-78
7	SOME COMMONLY USED SUBROUTINES	7-1
	MEMORY ADDRESSING	7-1
	AUTO INCREMENT AND AUTO DECREMENT	7-1
	INDEXED ADDRESSING	7-3
	INDIRECT ADDRESSING	7-3
	INDIRECT, POST-INDEXED ADDRESSING	7-4
	DATA MOVEMENT	7-5
	MOVING SIMPLE DATA BLOCKS	7-5
	MULTIPLE TABLE LOOKUPS	7-5
	SORTING DATA	7-7
	ARITHMETIC	7-8
	BINARY ADDITION	7-9

TABLE OF CONTENTS (Continued)

CHAPTER	PAGE
BINARY SUBTRACTION	7-10
DECIMAL ADDITION	7-11
DECIMAL SUBTRACTION	7-11
MULTIPLICATION AND DIVISION	7-12
8-BIT BINARY MULTIPLICATION	7-12
8-BIT BINARY DIVISION	7-15
16-BIT BINARY MULTIPLICATION	7-16
BINARY DIVISION	7-16
PROGRAM EXECUTION SEQUENCE LOGIC	7-17
THE JUMP TABLE	7-17
APPENDIX	
A	STANDARD CHARACTER CODES
	A-1

LIST OF FIGURES

FIGURE		PAGE
3-1	Printwheel Control Logic	Following 3-6
3-2	Printwheel Control Logic Timing Diagram	3-4
3-3	The Complete Simulation Program	3-65
4-1	Timing For Figure 3-1, From the Programmer's Viewpoint	4-2
4-2	Microcomputer Configuration	4-5
4-3	First Attempt At Program Flowchart	4-8
4-4	Program Flowchart To Compute Printhead Firing Pulse Length	4-16
4-5	A Simple Print Cycle Instruction Sequence Without Initialization Or Reset	4-18
4-6	A Simple Print Cycle Program	4-28
5-1	Microcomputer Configuration With A Single Interrupt	5-26
5-2	Generation Of The Restart Instruction Code Following An Interrupt For An 8080 Microcomputer System	5-36

LIST OF TABLES

TABLE		PAGE
2-1	Comparing Memory Utilization And Program Execution Speed For 7411 AND Gates' Simulation	2-26
5-1	The Shortest Economic Subroutine Length As A Function Of The Number Of Times The Subroutine Is Called	5-17
6-1	A Summary Of 8080/9080 Microcomputer Instruction Set	6-3
6-2	A Summary Of Instruction Object Codes And Execution Cycles	6-11

QUICK INDEX

INDEX	PAGE
A	ADDRESS BUS DECODE 3-10 AMPLIFIER 2-13 ASSEMBLY LANGUAGE VERSUS DIGITAL LOGIC 3-64 ASYNCHRONOUS LOGIC 2-11
B	BIT DATA 2-5,2-6 BIT MASKING 2-10 BIT SET/RESET ILLUSTRATED 5-7 BIT SET/RESET INSTRUCTIONS 5-5 BUFFER 2-13
C	CARRY STATUS 3-25 CH RDY 3-5 CHIP SELECT IN LARGER SYSTEMS 4-6 CHIP SELECT IN SIMPLE SYSTEMS 4-6 CLOCK SIGNAL 2-28 COMBINATORIAL LOGIC 1-1 COMPARE IMMEDIATE 4-25 COMPLEMENTING A BYTE OF MEMORY 2-15 CONDITIONAL INSTRUCTION EXECUTION PATHS 4-26 CONDITIONAL RETURN 5-18 CONFLICTS IN CPU REGISTER UTILIZATION 2-24 CPU REGISTERS 2-5
D	D TYPE FLIP-FLOP 2-28 DATA MEMORY ADDRESS COMPUTATION 3-56 DATA SOURCE AND DESTINATION 2-5 DIGITAL LOGIC DESIGN CYCLE 2-1 DIRECT VERSUS IMPLIED ADDRESSING 2-26
E	8212 I/O PORT USED IN INTERRUPT SYSTEM 5-28 EXECUTING PROGRAMS WITHIN TIME DELAYS 2-37 EXTERNAL LOGIC AS THE SOURCE OR DESTINATION 2-7 EVENT SEQUENCE 3-53 EVENT TIMING IN MICROCOMPUTER SYSTEM 3-26
F	FAN IN 2-14 FAN IN IN MICROCOMPUTER PROGRAMS 2-17 FAN OUT 2-14 FAN OUT IN MICROCOMPUTER PROGRAMS 2-19 FFA 3-7 FLIP-FLOP CLEAR 2-29 FLIP-FLOP PRESET 2-29 FLIP-FLOP SIMULATION USING I/O PORTS 3-23 FLOW CHART 2-5
G	GATE SETTLING TIME 2-12
H	H IN OPERAND FIELD 2-11 HARDWARE DEPENDENT INSTRUCTIONS 5-6

QUICK INDEX (Continued)

INDEX		PAGE
	HIGHER LEVEL LANGUAGES	4-4
I	IMPLIED ADDRESSING	2-25
	INPUT/OUTPUT	2-7
	INPUT SIGNAL PULSE WIDTH	3-14
	INPUT SIGNALS	4-1
	INTERRUPT ACKNOWLEDGE	5-27
	INTERRUPT ECONOMICS	5-34
	INTERRUPT ENABLE	5-27
	INTERRUPT PROGRAM ORIGIN	5-30
	INTERRUPT SERVICE TIME LOSS	5-34
	INTERRUPT TIMING CONSIDERATIONS	5-34
	INVERTER SIMULATION	3-24
	I/O IN MEMORY ADDRESS SPACE	2-7
	I/O PORT ADDRESS DETERMINATION	3-10
	I/O PORT ADDRESSING	3-10
	I/O PORT MODE SELECT INSTRUCTION SEQUENCE	3-11
	I/O PORT MODE SELECTION	3-10
	I/O PORT MODES	3-8
	I/O PORT PIN SELECT	2-10
	I/O PORT SELECT	4-4
	I/O PORTS ADDRESSED AS MEMORY	3-12
	I/O VIA I/O PORTS	2-8
J	JK FLIP-FLOP	2-28
	JUMP ON CONDITION	4-25
	JUMP ON NO CARRY	3-45
L	LATCHED BUFFER	3-8
	LEAKAGE CURRENT	2-14
	LIMIT CHECKING	4-23
	LOADING ADDRESS INTO STACK POINTER	7-2
	LOGIC EXCLUDED FROM MICROCOMPUTER	3-51
M	MACRO ASSEMBLER DIRECTIVE	5-23
	MACRO DEFINITION	5-23
	MACRO DEFINITION LOCATION IN A SOURCE PROGRAM	5-24
	MASTER-SLAVE FLIP-FLOP	2-31
	MASTER-SLAVE FLIP FLOPS	2-34
	MEMORY ADDRESSES	4-7
	MICROCOMPUTER LOGIC DESIGN CYCLE	2-2
	MONOSTABLE MULTIVIBRATOR	2-31
N	NEGATIVE EDGE TRIGGER	2-28
	NESTED SUBROUTINES	5-18
O	OBJECT CODE INTERPRETATION	2-8.2-10
	OBJECT PROGRAM	2-4
	ONE-SHOT	2-31
	ONE-SHOT INITIATION	2-39

QUICK INDEX (Continued)

INDEX	PAGE
	ONE-SHOT TIME DELAY SIMULATION 3-46
	ONE-SHOT TIME OUT USING STATUS 2-40
	ONE-SHOT VARIABLE PULSE 3-51
	OR GATE SIMULATION 3-24
P	PIN ASSIGNMENTS 4-3
	POSITIVE EDGE TRIGGER 2-27
	PPI CONTROL CODE 5-8
	PRINTHAMMER FIRING DELAY 4-15
	PRINTWHEEL POSITION OF VISIBILITY 3-7
	PRINTWHEEL READY 3-5
	PRINTWHEEL REPOSITIONING PRINT CYCLE 3-13, 3-30
	PROGRAM IMPLEMENTATION SEQUENCE 4-6
	PROGRAM TIMING 2-5
	PROGRAM VARIATIONS RANKED 2-27
	PROGRAMMABLE PERIPHERAL INTERFACE 3-8
	PROGRAMMED SIGNAL PULSE 4-20
	PROGRAMS MADE SHORTER 3-39, 3-42
	PULSE WIDTH CALCULATION 3-40
	PW STROBE 3-5
R	RAM 4-7
	RESET 3-23
	RESTART INSTRUCTION 5-27
	RESET LOGIC 4-6
	RESET THE CPU 3-15
	RESTORING STACK POINTER 7-2
	ROM 4-7
	ROM SELECT IN SIMPLE SYSTEMS 4-7
	RST INSTRUCTION CODE CREATION 5-28
S	SAVING REGISTERS AND STATUS 5-32
	SAVING STACK POINTER 7-2
	SETTLING DELAYS 3-6
	7474 FLIP-FLOP 3-21
	SIGNAL BUFFERING 2-14
	SIGNAL ENABLE 3-52
	SIGNAL LEVEL CHANGES SENSED WITHOUT INTERRUPTS 3-26
	SIGNAL PULSE WIDTH 3-15
	SIMPLE I/O 3-9
	SIMULTANEOUS TIME DELAYS 2-39
	SINGLE INTERRUPT CONFIGURATION 5-28
	SORTING DATA 7-7
	SOURCE PROGRAM 2-4
	SOURCE PROGRAM LABEL ASSIGNMENTS 2-21
	STACK MANIPULATION 5-21
	STACK POINTER MEMORY ADDRESSING 7-1
	STACK TOP 5-30

QUICK INDEX (Continued)

INDEX	PAGE
START RIBBON PULSE	3-7
STATUS CHANGES WITH INSTRUCTION EXECUTION	6-2
STATUS CONDITIONS	6-57
STATUS DETERMINATION BY ANDING A REGISTER WITH ITSELF	2-18
STATUS FLAGS USED TO REPRESENT LOGIC	3-24
STATUS TESTING USING DCX INSTRUCTION	2-36
STROBED I/O	3-9
SUBROUTINE CALL USING RST	6-65
SUBROUTINE PARAMETER	5-18
SWITCHING A BIT ON	3-26
SWITCHING BITS OFF	3-31
SWITCHING BITS ON	3-31
SYNCHRONOUS LOGIC	2-11
T	
TABLES POSITIONED TO SIMPLIFY ACCESS	
INSTRUCTION SEQUENCE	5-4
TIME DELAY	3-58
TIME DELAY BASED ON INPUT SIGNAL	3-16
TIME DELAY COMPUTATION	3-58
TIME DELAY INITIATION	2-37
TIME DELAY OF VARIABLE LENGTH	3-44, 4-20
TIMING AND LIMITS OF SIMULATION	3-42
TIMING AND LOGIC SEQUENCE	3-27, 3-34
	3-38
TIMING LONG TIME INTERVALS	2-36
TIMING SHORT TIME INTERVALS	2-35
TRANSFER FUNCTION	4-1
TTL LOADS	2-14
U	
USING IMPLIED MEMORY ADDRESSING	2-22
W	
WHEN TO USE INTERRUPTS	5-27
Z	
ZERO STATUS	3-25

Chapter 1

INTRODUCTION

This book explains how an assembly language program within a microcomputer system can replace combinatorial logic — that is, the combined use of "off-the-shelf", non-programmable logic devices, such as standard 7400 series digital logic.

**COMBINATORIAL
LOGIC**

If you are a logic designer, this book will teach you how to do your old job in a new way — by creating assembly language programs within a microcomputer system.

If you are a programmer, this book will show you how programming has found a new purpose — in logic design.

This is a "how to do it" book; as such, it has to become very specific, so a particular type of microcomputer, the 8080A, is referenced directly. A number of companies build 8080A type microcomputers; specifically, these products are covered:

**AMD 9080A
INTEL 8080A
NEC 8080A
TMS 8080A
NS 8080A**

Companies manufacturing these microcomputers are:

INTEL CORPORATION
3065 Bowers Avenue
Santa Clara, California 95051
ADVANCED MICRO DEVICES
901 Thompson Place
Sunnyvale, California 94086

TEXAS INSTRUMENTS, INC
P.O. Box 1444
Houston, Texas 77001

NEC MICROCOMPUTERS, INC
5 Militia Drive
Lexington, Massachusetts 02173

NATIONAL SEMICONDUCTOR CORP.
2900 Semiconductor Drive
Santa Clara, California 95050

WHAT THIS BOOK ASSUMES YOU KNOW

This book is a sequel to "An Introduction To Microcomputers", which was a single volume in its first edition, but is two volumes in its second edition.

"An Introduction To Microcomputers" describes microprocessors and microcomputers conceptually; it does not address itself to the practical matter of implementing a concept. This book addresses the practical matter of implementation.

In that this book is a sequel, it makes a single assumption — that you have read, or you otherwise understand the material covered in "An Introduction To Microcomputers". However, before launching into a real design project, you will need vendor literature that specifically describes the devices you have elected to use.

Note in particular that hardware and timing are not described in this book, either for the 8080A/9080A CPU, or any other microcomputer devices; sufficient information may be found in "An Introduction To Microcomputers", Volume II — Some Real Products.

The 8080A/9080A instruction set is described in Chapter 6 of this book, since programming is what this book is all about.

UNDERSTANDING ASSEMBLY LANGUAGE

Assembly language instructions are the transfer functions of a microcomputer system; taken together, they constitute an "instruction set", which describes the individual operations which the microcomputer can perform.

You define the events which must occur within the microcomputer system serially — as a sequence of instructions, which, taken together, constitute an assembly language program.

In reality, understanding what individual instructions do within a microcomputer system is very straightforward; it is one of the simplest aspects of working with microcomputers. Yet it unduly terrifies users who are new to programming. If that includes you, a word of advice — forget about mnemonics and instruction sets; take instructions one at a time as you encounter them in this book. When you do not understand what an instruction is doing, look it up in Chapter 6.

The specter of "programming" will haunt you only if you let it.

HOW THIS BOOK HAS BEEN PRINTED

Notice that text in this book has been printed in **boldface type** and lightface type. This has been done to help you skip those parts of the book that cover subject matter with which you are familiar. You can be sure that lightface type only expands on information presented in the previous boldface type. Therefore, only read boldface type until you reach a subject about which you want to know more at which point start reading the lightface type.

Chapter 2

ASSEMBLY LANGUAGE AND DIGITAL LOGIC

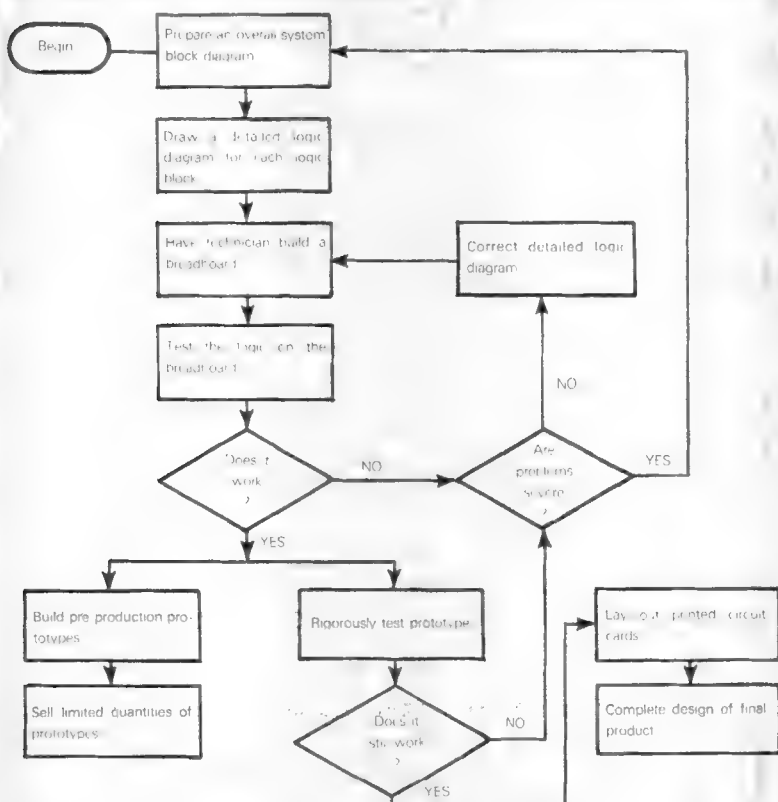
THE DESIGN CYCLE

Any product that is to be built out of discrete digital logic components will go through a well defined design cycle.

**DIGITAL
LOGIC
DESIGN
CYCLE**

Let us assume that the product has been defined — from marketing management's point of view.

You are presented with a product specification which identifies necessary product performance and characteristics, your job is to deliver a viable design to manufacturing. **The design cycle will proceed as follows:**



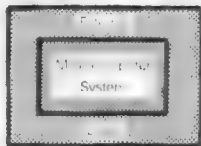
There is an expensive and slow iterative loop in any digital logic design cycle; as illustrated above, it consists of these steps:

- Redraw logic
- Build a new breadboard
- Test the breadboard for logic errors, technician errors or faulty components

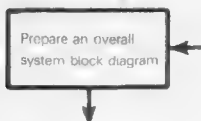
This iterative loop makes combinational logic design slow and expensive — not only during the initial design phase, but even more so when you subsequently decide to modify or enhance the product.

What happens when you start using microcomputers? First of all, a portion of your logic vanishes into a "black box" — which is the microcomputer system;

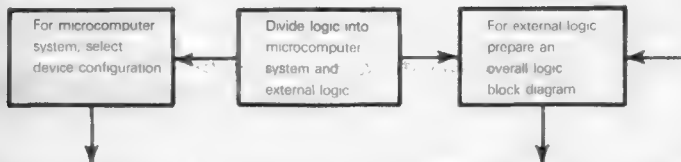
MICROCOMPUTER LOGIC DESIGN CYCLE



Your first step:



must now be broken out as follows:

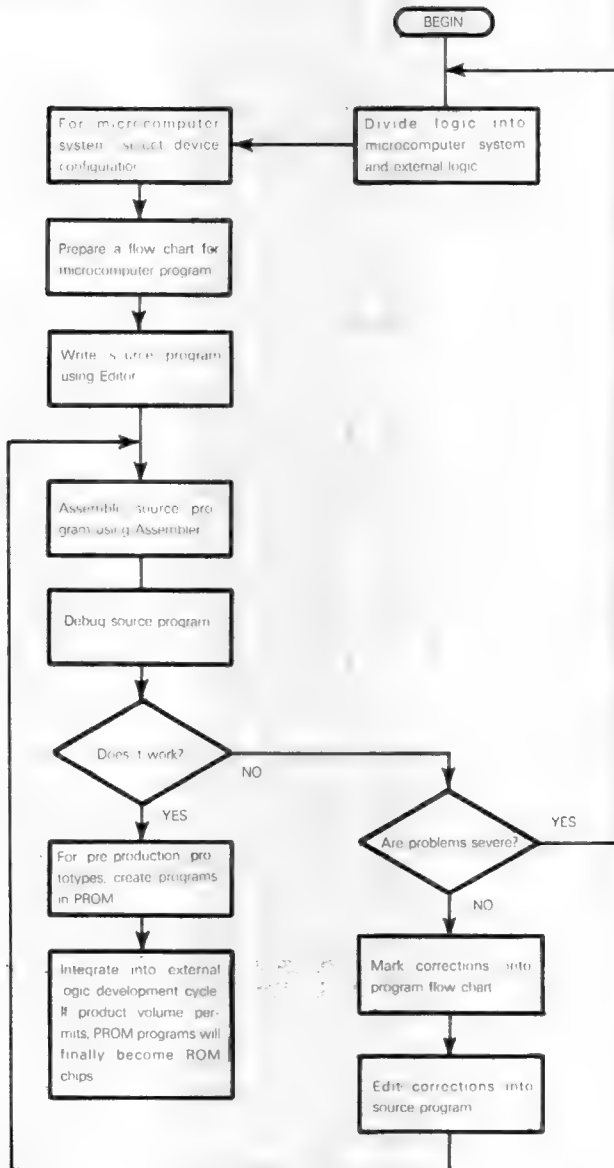


Partitioning your application into a microcomputer system and external digital logic may look like a difficult proposition — if you do not understand what the microcomputer system can do.

In fact, **once you have a microcomputer in your product, economics overwhelmingly favor making the "black box" assume as many tasks as possible; you must justify the existence of every single external logic gate.**

Remember, memory comes in finite increments. In order to expand the logic implemented within the microcomputer system, you may simply have to write additional instruction sequences that will reside in memory which would otherwise be wasted, adding program memory, for that matter, costs very little.

Also, compared to the cost of digital logic development, microcomputer logic development is quick and inexpensive. **A typical microcomputer system development cycle may be illustrated as follows:**



There are still iterative loops in the microcomputer development cycle illustrated above, but compared to digital logic development, less time and expense is associated with microcomputer development cycle iterative loops

Every microcomputer is supported by a development system. Characteristics and operation of these development systems vary markedly from one company to the next, however they **all have these capabilities:**

- 1) You can **simulate the microcomputer system** you have configured without necessarily creating a breadboard.
- 2) You can execute a resident editor program to **create your source program**. Remember, a sequence of assembly language instructions is referred to as a "Source Program".
- 3) You can **assemble the source program** right at the development system to create an object program. Remember, the source program becomes a sequence of binary digits, referred to as an object program, before it can be executed.
- 4) You can **conditionally execute the object program** to make sure that it works.

**SOURCE
PROGRAM**

**OBJECT
PROGRAM**

Using a typical microcomputer development system, you can go through several major development cycles in a single day, where each development cycle might have taken one or two weeks in a total digital logic implementation. Within a single development cycle you can make many program corrections, in less than a minute you can make a simple correction, equivalent to adding or removing a gate (or MSI function) from a digital logic breadboard.

SIMULATING DIGITAL LOGIC

OK, so logic must eventually be separated into a microcomputer system, and logic beyond the microcomputer system.

We are going to have to address two aspects of this logic separation:

- 1) Based on the ability of assembly language to simulate digital logic, **we must develop some simple criterion for estimating what a microcomputer system can do** and what it cannot do
- 2) **We must create a program to implement the logic functions which have been assigned to the microcomputer system.** Unfortunately, there are innumerable ways of writing a microcomputer program. Once you have mastered the concept of using instructions to drive a microcomputer system, **the next step is to learn how to write efficient programs.**

We will begin by describing simple digital logic simulation. This is a necessary beginning because there are some fundamental conceptual differences between digital logic and microcomputer programming logic.

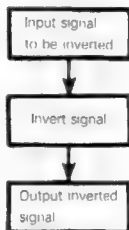
MICROCOMPUTER SIMULATION OF A SIGNAL INVERTER

Suppose you want to invert a single signal:



In the interests of developing good habits from the start, we will illustrate the signal inverter with the following logic flow chart:

**FLOW
CHART**



Although you would never use a microcomputer simply to replace a signal inverter, it is still worthwhile examining how it could be done.

A MICROCOMPUTER EVENT SEQUENCE

Recall that 8080 type microcomputers have the following CPU registers:

**CPU
REGISTERS**

	A	Primary Accumulator
B	C	Secondary Accumulators/Data Counter
D	E	Secondary Accumulators/Data Counter
H	L	Secondary Accumulators/Data Counter
SP		Stack Pointer
PC		Program Counter

This single instruction:

CMA :COMPLEMENT ACCUMULATOR

when converted into object code and executed, inverts all eight bits of the primary Accumulator. But that does not duplicate the inverter. First, one binary digit of the Accumulator must be selected to represent the signal being inverted. But which one?

**BIT
DATA**

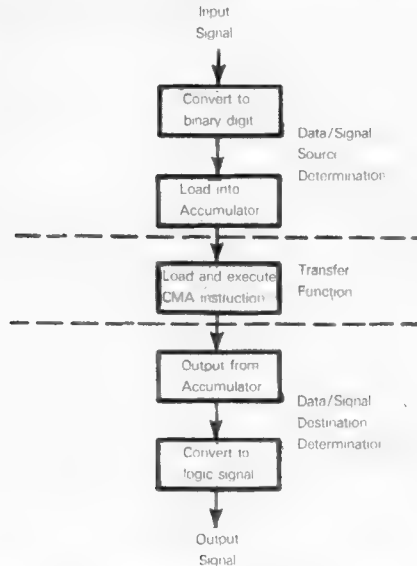
Having decided which binary digit, how does it reach the Accumulator in the first place? And once inverted, how does the inverted bit become a signal again?

If the CMA instruction object code must be executed in order to perform the actual inversion, how and when does the object code reach the CPU? Clearly execution of this instruction must be timed to occur after the binary digit to be inverted has reached the Accumulator.

**DATA SOURCE
AND
DESTINATION**

**PROGRAM
TIMING**

Steps needed to implement an inverter using a microcomputer may be illustrated by expanding our flow chart as follows:



In the illustration above, pay most attention to the division of the problem into these three phases:

- 1) **Data/signal source determination.** We identify the data which is to be operated on. This data is transferred to a location out of which it can be accessed by the microcomputer Central Processing Unit (CPU).
- 2) **Transfer function execution.** The actual operation which must be performed on the source data will be referred to as a "Transfer Function".
- 3) **Data/signal destination determination.** The data or signals having been subject to the transfer function, must now be transferred to some destination.

We will now generate an instruction sequence to implement the three phases of the inverter simulation illustrated above.

IMPLEMENTING THE TRANSFER FUNCTION

The CMA instruction inverts every bit of the Accumulator.

**BIT
DATA**

The CMA instruction therefore does not specify which bit of the Accumulator represents the signal to be inverted. This specification is implied by the way in which data is input to, and output from the microcomputer system.

DETERMINING DATA SOURCES AND DESTINATIONS

How will Accumulator data be input to, and output from the microcomputer system? In answering this question, we touch on one of the fundamental strengths (and complexities) of microcomputers — their flexibility.

The input signal and the inverted output signal are just what their names imply — they are signals. But to the microcomputer system, they are "external logic". Information transfers between external logic and the microcomputer system are referred to generically as Input/Output (or I/O).

**EXTERNAL LOGIC
AS THE SOURCE
OR DESTINATION**

During any programmed I/O operation, recall that the microcomputer is master and external logic is slave. This means that the microcomputer must indicate the direction of the I/O operation (input or output), and must identify the external logic being accessed

INPUT/OUTPUT

External logic might decode a specific memory address as an enable strobe, so that I/O is handled as though it were a memory read or write.

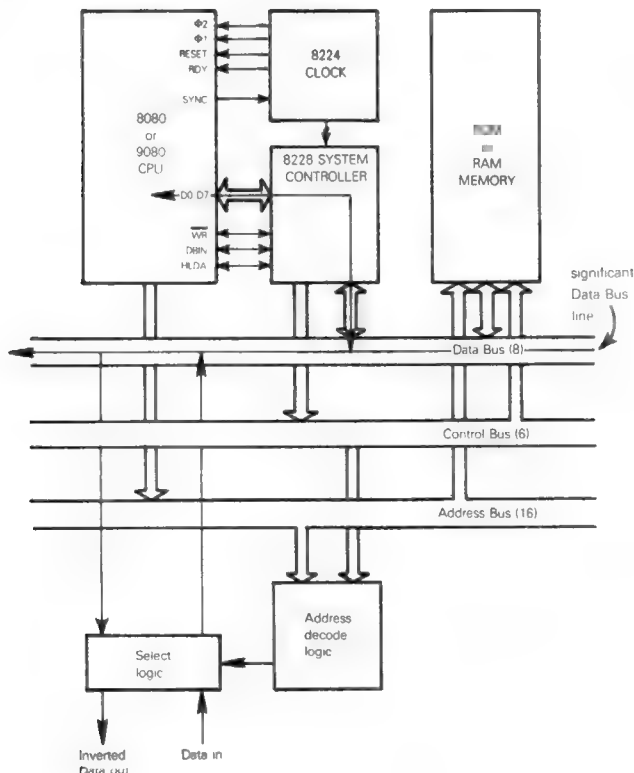
Suppose the label INVD is being used in the assembly language source program to identify the signal being inverted.

**I/O IN
MEMORY
ADDRESS
SPACE**

This is the instruction sequence which will reproduce the signal inverter:

LDA	INVD	;LOAD ACCUMULATOR FROM INVD
CMA		;COMPLEMENT THE ACCUMULATOR
STA	INVD	;STORE ACCUMULATOR CONTENTS AT INVD

In terms of microcomputer devices, this is the microcomputer configuration implied:



When the LDA instruction is executed, "Address Decode Logic" causes "Select Logic" to transmit the "Data In" signal to the Data Bus.

There are eight Data Bus lines; the number of the line to which the "Data In" signal is connected becomes the significant bit number within the Accumulator. When the LDA instruction has completed execution, the contents of the Data Bus will be in the Accumulator.

Next the CMA instruction is executed. This instruction causes every bit of the Accumulator to be complemented.

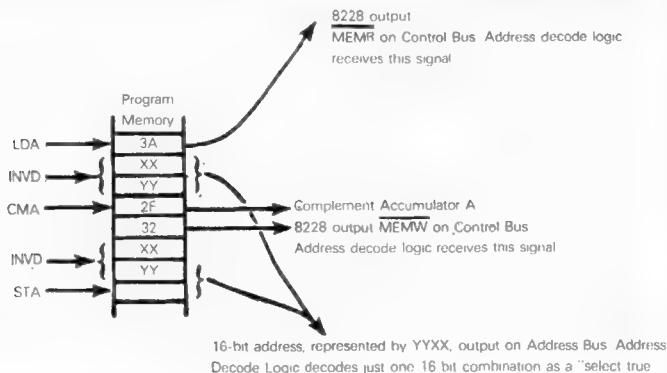
When the STA instruction is executed, the contents of the Accumulator are output to the Data Bus. "Address Decode Logic" then causes "Select Logic" to output the contents of a single Data Bus line — which becomes the inverted "Data Out" signal.

Because the "Select Logic" has "Data In" and "Data Out" signals connected to the same line of the Data Bus, "Data Out" is the complement of "Data In"; and the signal inverter has been simulated.

ROM or RAM memory must be present in the microcomputer system, because the object codes for the three instructions must be stored in, and fetched out of memory.

Consider object code in detail. The three source program instructions become object code as follows:

OBJECT CODE INTERPRETATION



The program memory addresses of the bytes within which the object codes are stored are not important. However, no memory byte, ROM or RAM, can have the address represented by YYXX, since external logic is selected by this address.

Observe that the two bytes of the 16-bit address YYXX are reversed when stored in memory. There is nothing very significant about this inversion, it is just the way 8080 devices were designed.

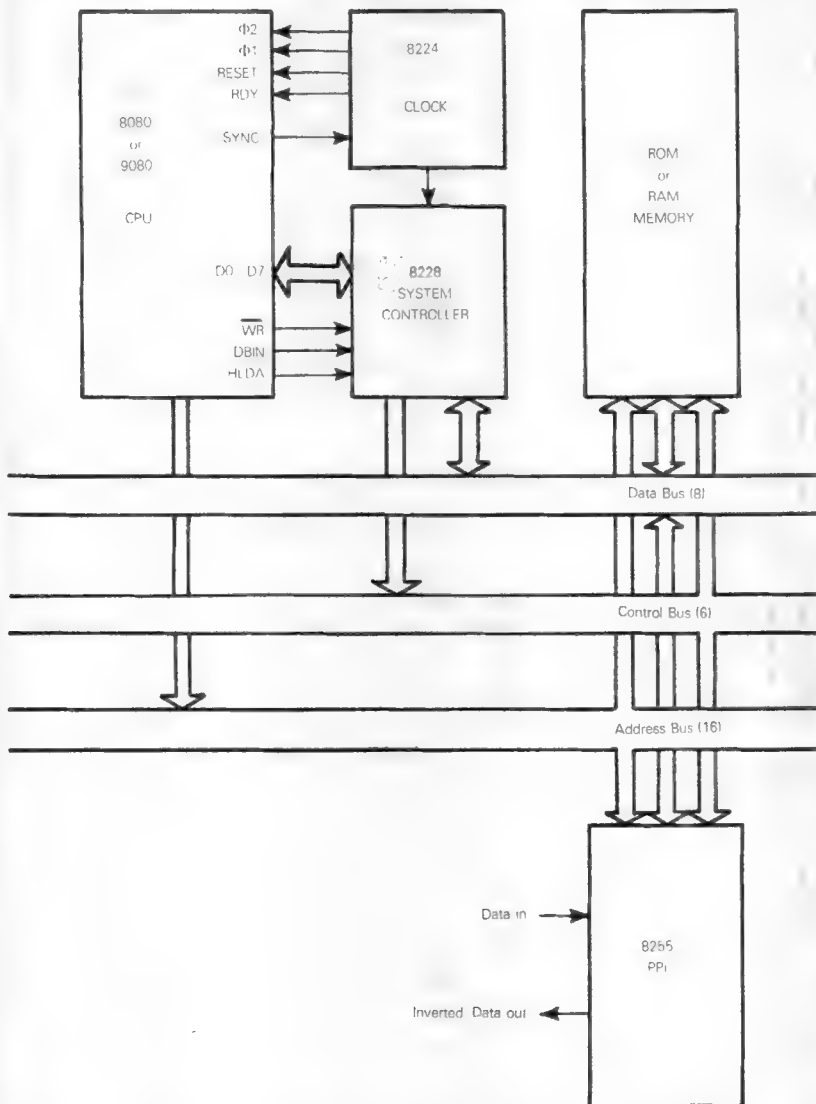
Now suppose that communication with external logic occurs via an I/O peripheral interface device.

I/O VIA I/O PORTS

In assembly language source program instructions, the label INVD will now identify an I/O port. This is the instruction sequence which reproduces the signal inverter:

IN	INVD	:INPUT TO ACCUMULATOR FROM PORT INVD
CMA		:COMPLEMENT THE ACCUMULATOR
OUT	INVD	:OUTPUT ACCUMULATOR TO PORT INVD

In terms of hardware, this is the microcomputer configuration implied:

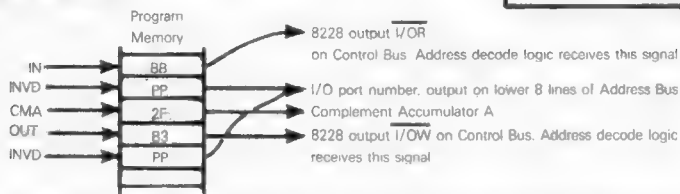


All we have done by adding the 8255 Programmable Peripheral Interface is provide the "Address Decode" and "Select Logic" needed by the "Data In" and inverted "Data Out" signals. Now the particular bit which is significant will be determined by the 8255 PPI pin to which the "Data In" and inverted "Data Out" signals are connected. In turn, these pins will be determined by the mode in which the 8255 PPI is used.

The fact that there are a number of options available to you when using the 8255 PPI is of no immediate consequence, in that it will confuse your early understanding of what assembly language programming is all about. We will therefore ignore 8255 PPI mode control instructions and simply assume that the appropriate mode control has been selected.

In this case the object code for the three instructions is interpreted as follows:

OBJECT CODE INTERPRETATION

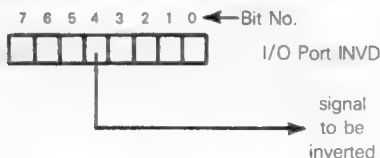


Once again, addresses of the program memory bytes within which the above object codes are stored will not be important.

Observe that we are complementing every bit of I/O port INVD, even though only one bit corresponds to the signal being inverted.

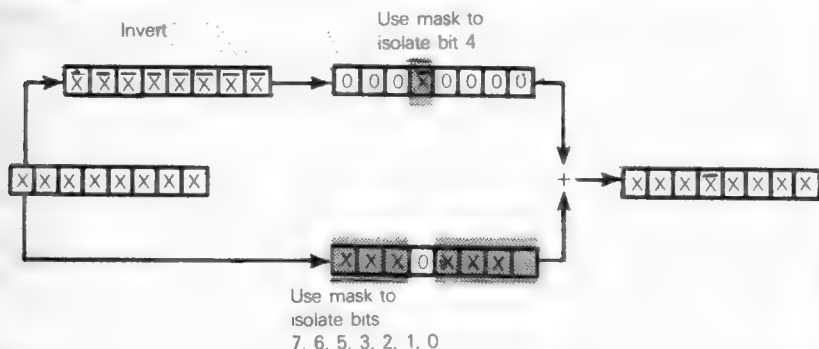
I/O PORT PIN SELECT

Suppose pin 4 alone must be inverted:



We can use a technique known as "masking" in order to invert a single I/O port pin, leaving all other pins alone. In this instance, masking may be illustrated as follows:

BIT MASKING



In the illustration above, X represents any binary digit; \bar{X} represents its complement.

The following instruction sequence will invert pin 4, leaving all other pins as they were:

IN	INVD	:INPUT TO ACCUMULATOR FROM I/O PORT INVD
CMA		:COMPLEMENT ACCUMULATOR
ANI	10H	:ISOLATE BIT 4
MOV	B,A	:SAVE IN REGISTER B

IN	INVD	INPUT TO ACCUMULATOR FROM I/O PORT INVD
ANI	EFH	CLEAR BIT 4
ORA	B	OR A AND B
OUT	INVD	OUTPUT ACCUMULATOR TO I/O PORT INVD

H as the last character in the operand field specifies a hexadecimal, immediate data value. Thus EFH represents the binary value:

**H IN
OPERAND
FIELD**

1 1 1 0 1 1 1 1
E F

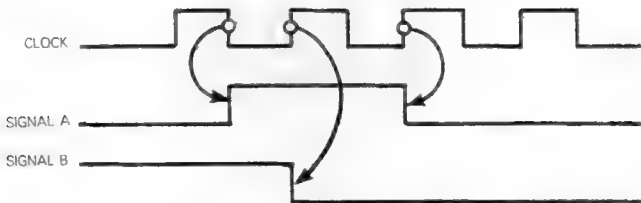
In terms of registers' contents, this is what happens when the above instruction sequence is executed (again X represents any binary digit):

		I/O Port	Accumulator	Register B
IN	INVD	XXXXXXXXXX	XXXXXXXXXX	?
CMA		XXXXXXXXXX	XXXXXXXXXX	?
			• 00010000	
ANI	10H	XXXXXXXXXX	000X0000	?
MOV	B,A	XXXXXXXXXX	000X0000	000X0000
IN	INVD	XXXXXXXXXX	XXXXXXXXXX	000X0000
			• 11101111	
ANI	EFH	XXXXXXXXXX	XX0XXXXX	000X0000
			+ 000X0000	
ORA	B	XXXXXXXXXX	XXXXXX	000X0000
OUT	INVD	XXXXXX	XXXXXX	000X0000

EVENT TIMING

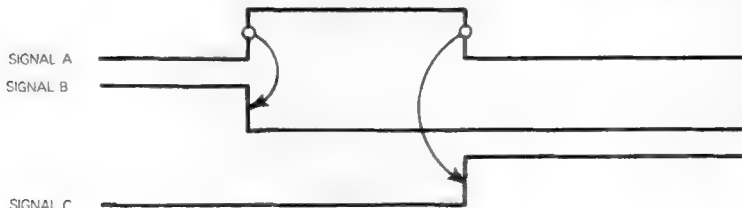
Within any digital logic implementation, events may be timed synchronously, based on a clock signal:

**SYNCHRONOUS
LOGIC**



or asynchronously, based upon an output signal from one device changing state and thus triggering another device's state change:

**ASYNCHRONOUS
LOGIC**



Simple gates, however, are continuous devices. Consider the following simple logic sequence:



The signal inverter continuously inverts its input; a gate settling time of perhaps 10 nanoseconds is the only lag between input and output signal state changes.

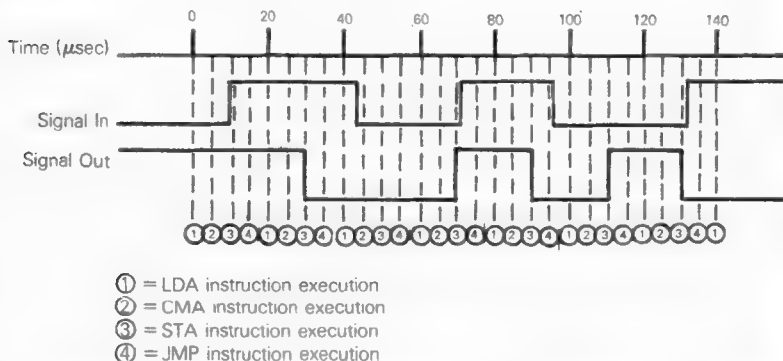
**GATE
SETTLING
TIME**

Within a microcomputer system, however, three instructions must be executed before an output signal can reflect an input signal's state change.

In the unlikely event that the microcomputer system is emulating an inverter and doing nothing else, the inverter instruction sequence could be continuously re-executed as follows:

LOOP	LDA	INVD	:LOAD ACCUMULATOR FROM INVD
	CMA		:COMPLEMENT THE ACCUMULATOR
	STA	INVD	:STORE ACCUMULATOR CONTENTS AT INVD
	JMP	LOOP	:RE-EXECUTE THE SIGNAL INVERTER SEQUENCE

Depending on the version of 8080-type microcomputer and the clock frequency, it will take approximately 20 microseconds to execute the signal inverter instruction loop once; providing the period between input signal state changes is never less than 20 microseconds, the microcomputer implemented signal inverter will always work. But **there may be a delay of up to 20 microseconds between an input signal changing state and the output signal following suit.** This may be illustrated as follows:



In the above illustration, the four instructions have been shown dividing twenty microseconds equally, so that each instruction is executed in five microseconds. In reality, this is not the case. Chapter 6 gives instruction execution times: you will see that the CMA instruction, for example, requires considerably less time to execute than any of the other three instructions. We will overlook this detail for the moment in order to concentrate on the concept at hand — which is that **we must pay careful attention to event sequences within the microcomputer system.**

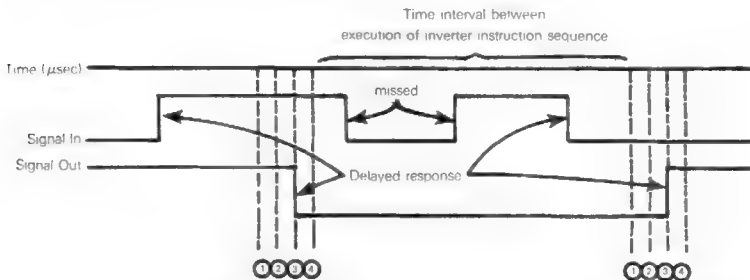
Irrespective of when and how "Signal In" changes state, it is the state of "Signal In" at time ① (when the LDA instruction is executed) which is transported, as a binary digit, into the microcomputer system.

The actual binary digit inversion occurs at time ②.

The inverted binary digit is converted into "Signal Out" at time ③, when the STA instruction is executed.

Thus, "Signal Out" timing may differ considerably from "Signal In" timing.

More serious problems arise when the signal inverter instruction sequence is just one small part of a larger microcomputer program. Under these circumstances, many milliseconds may elapse between repeated executions of the inverter instruction sequence. If you leave it to chance, signal inversions may be completely missed. At very best there may be considerable delays between the input signal changing state and the output signal following suit. This situation is illustrated as follows:



Again ①, ②, ③ and ④ identify LDA, CMA, STA and JMP instructions' execution, respectively.

Having stressed the importance of timing in a microcomputer system, plus the consequences of poor timing, we will drop the subject for the moment. This is because **timing problems largely evaporate when you simulate entire logic sequences as opposed to individual devices.** Therefore solutions to timing problems should be looked at in the context of an entire logic simulation; and we have not yet progressed that far.

BUFFERS, AMPLIFIERS AND SIGNAL LOADS

Having looked at timing, we will now turn to some other fundamental digital logic concepts

A signal buffer increases the signal current level:

BUFFER



An amplifier driver increases the signal voltage level:

AMPLIFIER



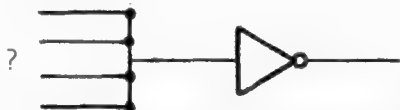
Every device has a well defined fan out. Fan out defines the number of parallel loads that may be connected to an output signal:

**FAN
OUT**



Logic devices will also have specified fan in, which indicates the number of parallel loads which may be connected to a device input:

FAN IN



What happens to these concepts once your logic disappears into a microcomputer program? The answer is simple: these concepts disappear — along with digital logic.

Now at the actual pins of a physical microcomputer device, fan in and fan out remain legitimate concepts; signals travelling between pins of individual microcomputer devices may need to be amplified and buffered. For example, an 8080-type CPU device's fan out may be as little as one or two Transistor-Transistor Logic (TTL) loads; that means if more than one or two similar devices connect to an output signal, the output signal will have insufficient power to transmit usable signals to all connected devices. Therefore for all but the simplest microcomputer configurations, bus lines will have to be buffered.

FAN IN
FAN OUT
TTL LOADS
SIGNAL BUFFERING

When determining whether your bus lines need to be buffered, do not ignore leakage current. For example, if you have sixteen ROM devices connected to the system bus, and only one device can be selected (and therefore connected) at any time, do not assume that the total signal load is due to the selected ROM. The fifteen unselected ROM devices will each tap off some leakage current; that alone may require system bus buffering.

**LEAKAGE
CURRENT**

Within a microcomputer program, however, when logic is totally represented by a microcomputer instruction sequence, you are dealing exclusively with binary digits — never with voltage or current levels. Fan in is infinite, since the status of a binary digit may be the result of any number of logical computations. Fan out is infinite, since you can read the status of a binary digit as often as you want. Buffers and amplifiers are meaningless, since a binary digit has no qualities equivalent to voltage or current. A binary digit offers pure, finite resolution.

Take another look at the signal inverter, as simulated by a microcomputer.

We will take a giant conceptual step and assume that the signal inverter is buried within a logic sequence, such that no input or output signal is generated at any microcomputer device pin. In other words, the signal inverter becomes a small part of a larger transfer function.

The input to the signal inverter is a binary digit created by some previous logic.

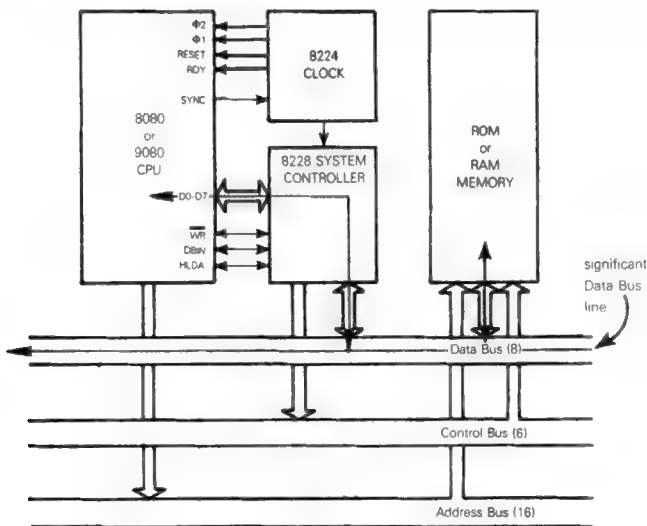
The output from the signal inverter is another binary digit which becomes input to subsequent logic.

Logic external to the microcomputer system does not supply the inverter input as a signal arriving at a microcomputer device pin, nor does the inverted signal get transmitted to external logic via a microcomputer device pin. Rather, the interface between external logic and the microcomputer system occurs at some point significantly before and beyond the signal inverter. **Our signal inverter may now be represented by these same three instructions:**

**COMPLEMENTING
A BYTE OF
MEMORY**

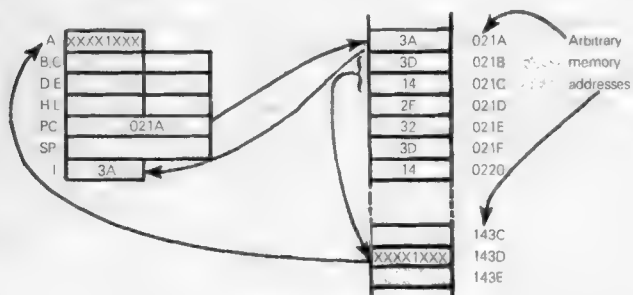
LDA	INVD	;LOAD ACCUMULATOR FROM INVD
CMA		;COMPLEMENT
STA	INVD	;STORE ACCUMULATOR CONTENTS AT INVD

The source and destination become data memory bits; this may be illustrated as follows:

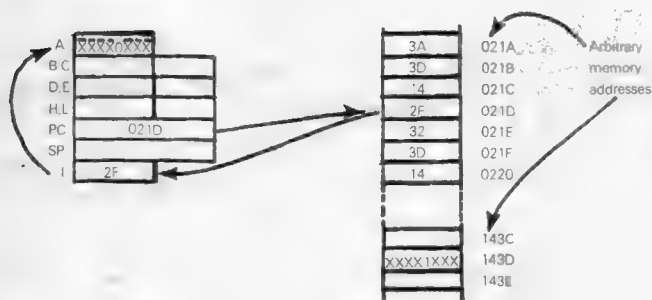


In terms of memory and CPU register contents, the signal inverter sequence proceeds as follows:

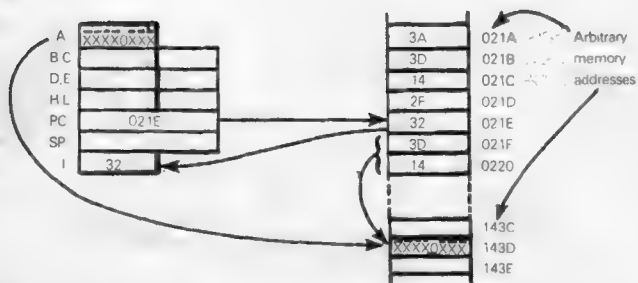
①



②



③



With regard to the above illustration, the letters A, B, C, D, E, H and L identify the seven CPU registers of 8080-type CPUs. PC represents the Program Counter. SP represents the Stack Pointer. I represents the Instruction register.

The contents of data memory byte $143D_{16}$ and the A register are represented in binary format. X represents any binary digit. Note that we have arbitrarily selected bit 3 to be the significant bit.

In step ①, the LDA instruction is executed. This instruction causes the contents of data memory byte $143D_{16}$ to be loaded into the Accumulator.

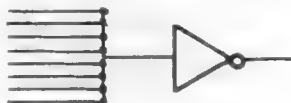
During step ②, the CMA instruction is executed. This causes the contents of the Accumulator to be complemented.

During step ③, the contents of the Accumulator are loaded back into memory byte 143D₁₆.

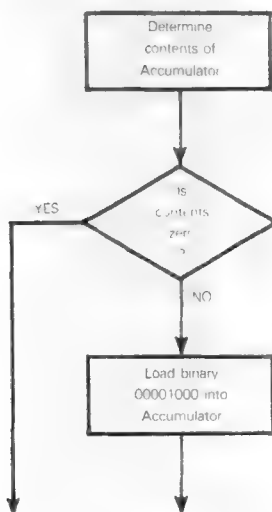
Signal inversion has been simulated by inverting the contents of bit 3 (along with every other bit) of data memory byte 143D₁₆.

Where does the inverter's input come from? A data memory bit. **Let us suppose, to illustrate a point, that the inverter input is the OR of eight signals.** We could not wire-OR these eight signals to create an inverter input as follows:

**FAN IN
IN MICRO-
COMPUTER
PROGRAMS**



But **presuming the eight signals are represented by the eight binary digit contents of the Accumulator**, we would have no trouble generating the inverter input via the following logic sequence:



The fan in logic is implemented by this instruction sequence:

;ASSUME THE EIGHT SIGNALS ARE IN THE ACCUMULATOR.

;EACH REPRESENTED BY ONE ACCUMULATOR BIT

ANA	A	;	AND ACCUMULATOR WITH ITSELF TO SET STATUS FLAGS
JZ	NEXT	;	ACCUMULATOR HOLDS 0. SIGNAL IN MUST BE 0
MVI	A,8	;	ACCUMULATOR HOLDS NONZERO. SIGNAL IN MUST BE 1
NEXT	STA	INVD	;
			SAVE INVERTER INPUT

The above instruction sequence is a direct microcomputer program implementation of the eight signal wire-OR. Let us examine how the instruction logic works.

We are going to assume that the eight input signals are initially represented by the status of the eight Accumulator binary digits:



We are further going to assume that, in keeping with the prior illustration, bit 3 of the data byte will ultimately be the significant inverter signal bit.

Since the inverter input is the wire-OR of eight signals, program logic must set bit 3 of the Accumulator to 1 if any Accumulator bit is nonzero; bit 3 of the Accumulator must be set to 0 if all Accumulator bits are zero. The contents of the Accumulator are then stored in the data memory byte represented by label INVD. With regard to the previous illustration, INVD would be a label representing memory byte 143D₁₆.

This is how the four-instruction sequence illustrated above works:

We do not know what the Accumulator initially contains, so we must determine its contents by setting CPU status flags appropriately. To do this we AND the Accumulator contents with itself. ANDing the contents of the Accumulator with itself does not change the contents of the Accumulator, but status flags are set. We are only interested in the Zero status, which will be set to 1 if the AND of the Accumulator with itself generates a zero result, the Zero status flag will be set to 0 otherwise.

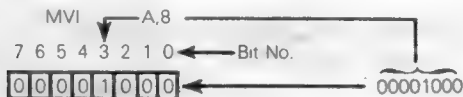
**STATUS
DETERMINATION
BY ANDING A
REGISTER WITH
ITSELF**

But the AND of the Accumulator with itself will only be zero if the Accumulator contains zero:

```
00000000
.....
00000000
.....
00000000
```

Thus after execution of the ANA instruction, if the Zero status is 1, then bit 3 of the Accumulator must already be 0, which is what we want it to be. No operation is required and we jump to the STA instruction.

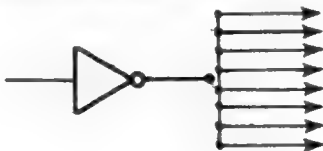
If the Zero bit was 0, then one or more bits of the Accumulator are nonzero. The MVI instruction loads a 1 into bit 3 of the Accumulator:



Finally the STA instruction is executed to load the inverter input signal into the appropriate data memory byte.

Now suppose the inverter output is distributed to numerous subsequent devices.

The following logic represents fan out that is not feasible:



Within a microcomputer program, the whole concept of fan out disappears. The inverter output may be accessed an indefinite number of times by the simple re-execution of an LDA instruction:

**FAN OUT
IN MICRO-
COMPUTER
PROGRAMS**

```
LDA    INVD    ;LOAD INVERTER OUTPUT INTO AC-
                CUMULATOR
.
.
.
LDA    INVD    ;LOAD INVERTER OUTPUT INTO AC-
                CUMULATOR
.
.
.
LDA    INVD    ;LOAD INVERTER OUTPUT INTO AC-
                CUMULATOR
.
.
.
LDA    INVD    ;LOAD INVERTER OUTPUT INTO AC-
                CUMULATOR
.
.
.
LDA    INVD    ;LOAD INVERTER OUTPUT INTO AC-
                CUMULATOR
```

What about amplifiers and buffers? Clearly within the context of binary data stored in memory, they have no meaning. If amplifiers and buffers are present because of the electrical characteristics of the memory and processor chips, that has nothing to do with the logic function being implemented by a microcomputer program

MICROCOMPUTER SIMULATION OF 7404/05/06/07 HEX INVERTERS

These four hex inverters differ only in their electrical characteristics:

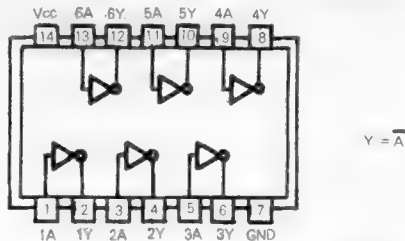
The 7404 is a simple hex inverter.

The 7405 is a hex inverter with open collector outputs.

The 7406 is a hex inverter buffer/driver with open collector, high voltage outputs.

Since these three devices differ only in their electrical characteristics, within a microcomputer assembly language simulation they are identical. Let us look at the

7404. It consists of six independent signal inverters, which may be illustrated as follows:



The instruction sequence to represent a hex inverter is identical to the three-instruction, single signal inverter instruction sequence, because 8080-type microcomputers are eight-bit parallel devices. Whether you like it or not, this inverter instruction sequence inverts eight independent binary digits. Hex inverters may therefore be represented within a microcomputer instruction sequence as follows:

```

LDA  INVD  ;LOAD ACCUMULATOR FROM INVD
CMA      ;COMPLEMENT
STA  INVD  ;STORE ACCUMULATOR CONTENTS TO INVD

```

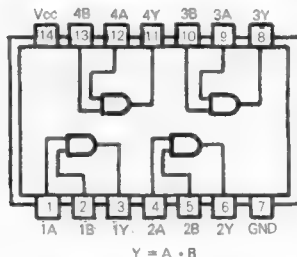
We will arbitrarily identify significant bits, as implied by the hex inverter, as follows:



Note that the above selection of significant bits is completely arbitrary. There is absolutely no practical or philosophical argument favoring any one bit assignment as compared to any other.

MICROCOMPUTER SIMULATION OF 7408/09 QUADRUPLE TWO-INPUT POSITIVE AND GATES

These two devices provide four independent, two-input, one output AND gates, which may be illustrated as follows:

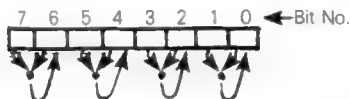


The 7409 has open collector outputs, which differentiates it from the 7408. This difference has no meaning in a microcomputer program simulation, therefore the two devices can be looked on as being identical.

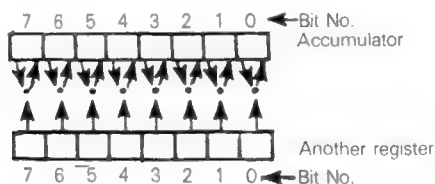
TWO INPUT FUNCTIONS

From the microcomputer programmer's point of view, the most significant difference between a 7408 AND gate and a 7404 inverter is not the logic function; rather it is the fact that a 7408 is a two-input device. Conceptually, we might imagine a 7404 being simulated in one of the two following ways:

- 1) The eight input signals are loaded into the CPU Accumulator register. Each even numbered bit is ANDed with the bit to its right. The result is deposited in the even-numbered bit for each bit pair:



- 2) The two sets of four inputs are loaded into the CPU Accumulator and one other register. The result is returned in the Accumulator:



Upon examining the 8080 microcomputer instruction set, you will find that the second method of simulating a 7408 is the natural one. This is the required instruction sequence:

LDA	SRCA	:LOAD FIRST SET OF INPUTS. FROM SRCA
MOV	B,A	:SAVE IN THE B REGISTER
LDA	SRCB	:LOAD SECOND SET OF INPUTS. FROM SRCB
ANA	B	:AND B WITH A
STA	DST	:SAVE RESULT IN DST

If the use of labels SRCA, SRCB and DST still confuse you, let us take a minute to clarify them. Eventually you will have some

amount of memory which may vary from as little as 256 bytes to as much as 65,536 bytes. Each of the labels SRCA, SRCB and DST identify one memory byte. At the time you are writing the source program, the exact memory byte identified by each label is unimportant. When you eventually assemble your source program, the assembler listing will print a memory map. The memory map will identify the exact memory byte associated with each label you have used. By examining the memory map, you will be able to determine whether or not all label assignments are valid. If any label assignments are invalid, you will have to take appropriate action. Appropriate action may involve adding more memory to your microcomputer configuration, or you may have to rewrite your source program, so that it makes more effective use of the memory you have.

**SOURCE
PROGRAM
LABEL
ASSIGNMENTS**

The problem of labels and memory allocations is irrelevant at the present level of discussion. Simply imagine every label as addressing one specific memory byte. Do not worry about which memory byte will eventually be addressed and your problem will disappear.

The 7408 simulation instruction sequence illustrated above by no means represents the only way in which a 7408 may be simulated.

First consider some minor variations. CPU registers C, D, E, H or L could be used instead of Register B to hold the second data input. Here is one example:

LDA	SRCA	;LOAD FIRST SET OF INPUTS, FROM SRCA
MOV	C,A	;SAVE IN THE C REGISTER
LDA	SRCB	;LOAD SECOND SET OF INPUTS, FROM SRCB
ANA	C	;AND C WITH A
STA	DST	;SAVE RESULT IN DST

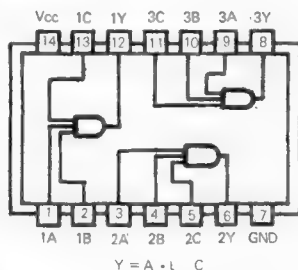
Using registers H or L to hold the second input is not encouraged. The primary use for these two registers is to hold a data memory address. For example, the LDA and STA instructions could be replaced as follows:

**USING
IMPLIED
MEMORY
ADDRESSING**

LXI	H,SCRA	;LOAD ADDRESS FOR FIRST SET OF INPUTS INTO H,L
MOV	A,M	;LOAD FIRST SET OF INPUTS INTO A
LXI	H,SCRB	;LOAD ADDRESS OF SECOND SET OF INPUTS INTO H,L
ANA	M	;AND SECOND SET OF INPUTS WITH A
LXI	H,DST	;LOAD ADDRESS OF DESTINATION INTO H,L
MOV	M,A	;STORE RESULT IN DST

THE MICROCOMPUTER SIMULATION OF A 7411 TRIPLE, THREE-INPUT, POSITIVE AND GATE

The principle difference between the 7411 AND gate and the 7408 AND gate is the number of input signals. The 7411 generates three output signals, each of which is the AND for three inputs:



THREE INPUT FUNCTIONS

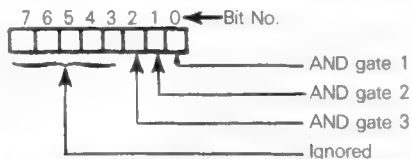
Again we are faced with choices. We may load the three sets of inputs into three CPU registers (the Accumulator and two other registers), then perform two ANDs before restoring the result:

ONE	LDA	SRCA	;LOAD FIRST SET OF INPUTS, FROM SCRA
TWO	MOV	B,A	;SAVE IN B REGISTER
THRE	LDA	SRCB	;LOAD SECOND SET OF INPUTS, FROM SRCB
FOUR	MOV	C,A	;SAVE IN C REGISTER
FIVE	LDA	SRCC	;LOAD THIRD SET OF INPUTS, FROM SCRC
SIX	ANA	B	;AND B WITH A
SEVN	ANA	C	;AND C WITH A
EIGT	STA	DST	;SAVE THE RESULT IN DST

The instructions in the above sequence have been given labels so as to make the description

which follows easier to understand. The instructions do not need labels in order to satisfy the needs of an assembly language source program.

When instruction ONE executes, an 8-bit value is loaded into the Accumulator from the memory byte addressed by label SCRA. We will assume that AND gate inputs are represented as follows:



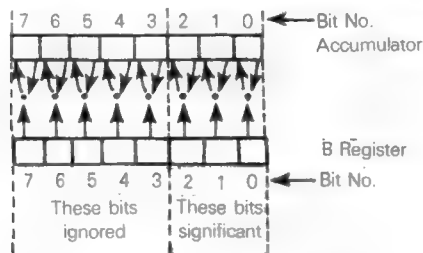
Understand that the assignment of data bits illustrated above is completely arbitrary. It is only necessary that all subsequent inputs be consistent.

After instruction ONE has executed, the first set of inputs is in the Accumulator. The Accumulator is the only CPU register into which data may be loaded if you use direct addressing. The first set of inputs must therefore be saved in another register, so that the Accumulator is free for a second set of inputs to be loaded. Instruction TWO moves the contents of the Accumulator to the B register.

Instructions THREE and FOUR load the second set of inputs into the Accumulator, then move it to the C register. We assume that bit assignments of this second set of inputs are identical to the bit assignments illustrated above for the first input.

The third and last set of inputs is loaded into the Accumulator by instruction FIVE.

The ANA instruction ANDs the contents of the CPU register with the contents of the Accumulator, leaving the result in the Accumulator. Instruction SIX performs the first AND as follows:



Instruction SEVN performs the second AND operation. This time the AND occurs between the Accumulator and Register C. The Accumulator initially holds the result of the AND with B, illustrated above. After instruction SEVN has executed, the AND of three inputs is in the Accumulator. Instruction EIGHT returns the final result to a memory byte addressed by the label DST. The 7411 AND gate simulation is complete.

Now consider an alternative simulation of the 7411 AND gates. We may load the first input into the Accumulator and the second input into another register. After ANDing these two inputs, we may load the third input into the same "other" register, AND it with the result of the first AND, then return the result:

ONE	LDA	SCRA	;LOAD FIRST SET OF INPUTS, FROM SCRA
TWO	MOV	B,A	;SAVE IN B REGISTER
THRE	LDA	SRCB	;LOAD SECOND SET OF INPUTS, FROM SRCB
FOUR	ANA	B	;AND B WITH A. THE RESULT IS IN A
FIVE	MOV	B,A	;SAVE THE RESULT IN B
SIX	LDA	SRCC	;LOAD THIRD SET OF INPUTS, FROM SRCC

SEVN . . . ANA . . . B . . . :AND B WITH A
 EIGT . . . STA . . . DST . . . :SAVE THE RESULT IN DST

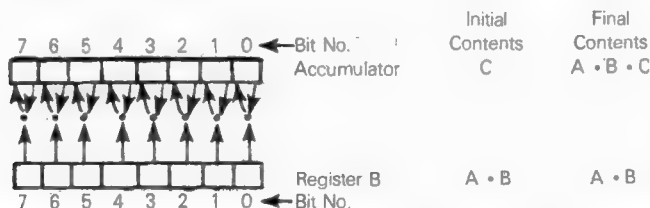
Let us compare this second simulation of the 7411 AND gate with the first simulation. Instructions ONE, TWO and THREE are identical to the first simulation. After these three instructions have executed, one set of inputs is in Register B and a second set of inputs is in the Accumulator. This is the situation:

Inputs A are in the Accumulator
 Inputs B are in Register B

Now instead of bringing the third set of inputs immediately into a CPU register, we execute instruction FOUR, which generates the AND of the first two inputs. Since this AND is generated in the Accumulator, we save the result in Register B by executing instruction FIVE. This is the net effect:

$A \cdot B$ in Register B

Now instruction SIX loads the third set of inputs into the Accumulator. Instruction SEVN ANDs the third set of inputs with the result of the first AND as follows:



Instruction EIGT saves the result from the Accumulator in the memory byte addressed by label DST.

MINIMIZING CPU REGISTER ACCESSES

Which is the "better" 7411 AND gates' simulation? Clearly the second option.

There is a nonobvious problem associated with the indiscriminate use of CPU registers. We have arbitrarily decided that Register B will hold a second input. So long as we are simulating 7411 AND gates, without regard to what precedes or follows, the selection of Register B is arbitrary; its selection carries no rewards or consequences.

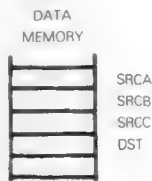
Invariably, an instruction sequence such as the 7411 AND gates' simulation is just a small part of a larger whole. Now we must worry about whether using Register B to house the second input will interfere with prior or subsequent use of Register B. A very common programming error involves CPU register utilization conflicts. For example, what if some prior logic step uses Register B to hold an intermediate data value? Now the 7411 simulation will wipe out the data which was being temporarily stored in this register.

**CONFLICTS
 IN CPU
 REGISTER
 UTILIZATION**

In order to reduce CPU register conflicts, it is always preferable to choose an instruction sequence that uses as few CPU registers as possible, providing there is no significant penalty. In this case there is no significant penalty. It takes no more instructions to simulate 7411 AND gates using CPU Register B only, than it does using CPU registers B and C. Using CPU Register B only is therefore the better method.

Now let us consider a 7411 AND gates' simulation using implied addressing. Assume that the three inputs to the AND gates are stored in sequential bytes of data memory and the destination follows the last source byte, as follows:

IMPLIED ADDRESSING



Now using implied addressing, we have the following instruction sequence:

ONE	LXI	H, SRC A	:LOAD THE FIRST SOURCE ADDRESS INTO HL
TWO	MOV	A, M	:LOAD THE FIRST SOURCE INTO THE ACCUMULATOR
THRE	INX	H	:INCREMENT THE IMPLIED ADDRESS
FOUR	ANA	M	:AND ACCUMULATOR WITH SECOND SOURCE
FIVE	INX	H	:INCREMENT THE IMPLIED ADDRESS
SIX	ANA	M	:AND ACCUMULATOR WITH THIRD SOURCE
SEVN	INX	H	:INCREMENT THE IMPLIED ADDRESS
EIGT	MOV	M, A	:SAVE THE RESULT

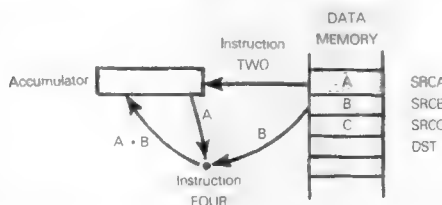
This is how the instruction sequence will be executed:

Instruction ONE loads the address of the first source byte into the H and L registers.

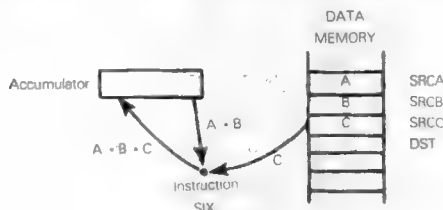
Instruction TWO moves the contents of the memory byte addressed by H and L into the Accumulator.

Instruction THREE increments the 16-bit address in the H and L registers, which now addresses SRCB.

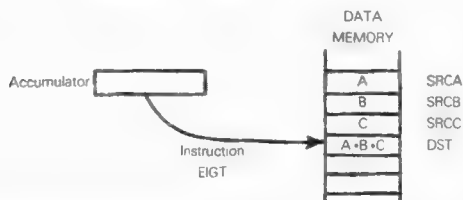
Instruction FOUR ANDs the contents of the Accumulator with the second source, as addressed by the H and L registers. The result is saved in the Accumulator. This may be illustrated as follows:



Instructions FIVE and SIX increment the implied address and repeat the AND operation, this time ANDing the third input with the AND of the first two inputs. This may be illustrated as follows:



The address in H and L is incremented again so that it now points to DST. Instruction EIGHT saves the result in the destination as follows:



COMPARING MEMORY UTILIZATION AND EXECUTION SPEED

We now have these three programs, all of which simulate 7411 AND gates:

Program 1 uses direct addressing and three CPU registers.

Program 2 uses direct addressing and two CPU registers.

Program 3 uses implied addressing.

Let us compare the number of object program bytes required to store each program, and the number of CPU clock cycles required to execute each program. The results are summarized in Table 2-1. Table 2-1 includes the instruction mnemonics for each program to help you follow how total object program bytes and execution cycles have been computed. See Chapter 6 for the data you need in order to verify Table 2-1.

PROGRAM 1			PROGRAM 2			PROGRAM 3		
MNEMONIC	BYTES	CYCLES	MNEMONIC	BYTES	CYCLES	MNEMONIC	BYTES	CYCLES
LDA	3	13	LDA	3	13	LXI	3	10
MOV ¹	1	5	MOV ¹	1	5	MOV ²	1	7
LDA	3	13	LDA	3	13	INX	1	5
MOV ¹	1	5	ANA ¹	1	5	ANA ²	1	7
LDA	3	13	MOV ¹	1	13	INX	1	5
ANA ¹	1	4	LDA	3	4	ANA ²	1	7
ANA	1	4	ANA ¹	1	4	INX	1	5
STA	3	13	STA	3	13	MOV ²	1	7
TOTAL	16	70	TOTAL	16	70	TOTAL	10	53
¹ Register-register version of instruction								
² Register-memory version of instruction								

Table 2-1. Comparing Memory Utilization And Program Execution Speed For 7411 AND Gates' Simulation

Programs 1 and 2 have identical memory utilization and execution speeds — which is not surprising, since they vary the sequence in which the same instructions are executed. **Program 3 adopts a completely different philosophy towards the 7411 AND gates' simulation by using implied memory addressing, rather than direct memory addressing. The result is dramatic. Six bytes of memory are saved and the program executes in 76% of the time.** But Program 3 places an additional restriction on the simulation: the three data sources and the destination must occupy four contiguous bytes of data memory.

**DIRECT
VERSUS
IMPLIED
ADDRESSING**

How are we going to rank the three simulation options?

**PROGRAM
VARIATIONS
RANKED**

We have already concluded that Program 2 beats Program 1, because Program 1 makes gratuitous use of an extra CPU register. Program 3 is clearly better than Program 2, providing the restriction on data source and destination locations is tolerable.

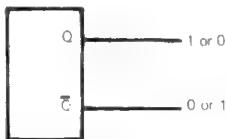
Regarding Program 3's superiority over Program 2, it is worth noting again, as was stressed in "An Introduction To Microcomputers", that the indiscriminate use of direct addressing in microcomputer applications can be costly. Implied memory addressing may appear primitive to a programmer with minicomputer or large computer background, but it is economical.

THE MICROCOMPUTER SIMULATION OF A 7474 DUAL, D-TYPE, POSITIVE EDGE TRIGGERED FLIP-FLOP WITH PRESET AND CLEAR

Before looking at the 7474 flip-flop in particular, let us consider flip-flops in general. First a few definitions.

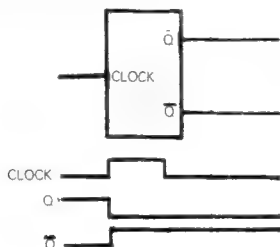
A DIGITAL LOGIC DESCRIPTION OF FLIP-FLOPS

A flip-flop is a bistable logic device, that is, a device which may exist in one of two stable conditions. 7474 type flip-flops have two outputs, Q and \bar{Q} ; thus the two bistable conditions may be represented as follows



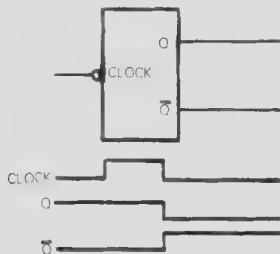
A clock signal causes the flip-flop to change from one bistable condition to the other. A positive edge triggered flip-flop changes upon sensing a zero-to-one transition of the clock signal:

**POSITIVE
EDGE
TRIGGER**



A negative edge triggered flip-flop changes state upon sensing a one-to-zero clock signal transition:

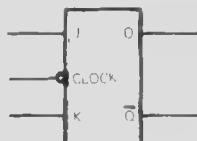
NEGATIVE EDGE TRIGGER



A JK flip-flop preconditions the Q and \bar{Q} outputs which will be generated by the next clock edge trigger as follows:

JK FLIP-FLOP

Status of J and K at clock signal		Outputs generated at clock signal	
J	K	Q	\bar{Q}
1	0	1	0
0	1	0	1
0	0	Stay as you were Change state regardless of previous state.	
1	1		



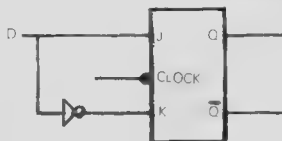
In the above table, "clock signal" will be a zero-to-one transition for a positive edge triggered device; it will be a one-to-zero transition for a negative edge triggered device. This definition of "clock signal" also applies to the D type flip-flop described next.

CLOCK SIGNAL

By inverting a J input in order to generate the K input, a D type flip-flop is created. These are the D type flip-flop characteristics that result:

D TYPE FLIP-FLOP

Status of J and K at clock signal		Outputs generated at clock signal	
J = D	K = \bar{J}	Q	\bar{Q}
1	0	1	0
0	1	0	1



Here is a positive edge triggered, D-type flip-flop timing diagram:



A D-type flip-flop therefore will always output the input conditions that existed at the previous clock pulse.

The presence of a Preset input means that the flip-flop may be forced to output $Q = 1$ and $\bar{Q} = 0$. Preset true forces this condition.

**FLIP-FLOP
PRESET**

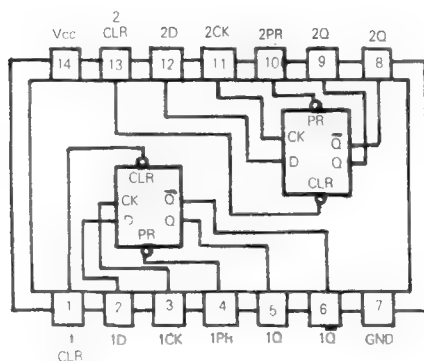
A Clear input is the opposite of a Preset input. When true, the Clear input forces $Q = 0$ and $\bar{Q} = 1$.

**FLIP-
FLOP
CLEAR**

Combining the definitions given above, this is what we get for a 7474 type flip-flop:

FUNCTION TABLE

INPUTS				OUTPUTS	
1PR or 2PR	1CLR or 2CLR	1CK or 2CK	1D or 2D	1Q or 2Q	1 \bar{Q} or 2 \bar{Q}
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H*	H*
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	\bar{Q}_0

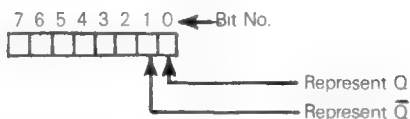


In the function table above, \uparrow represents a clock zero-to-one transition. H* signifies an unstable state. Q_0 is the previous state for Q. X signifies "Don't care".

AN ASSEMBLY LANGUAGE SIMULATION OF FLIP-FLOPS

Now our first problem, when trying to simulate a 7474 flip-flop, is the fact that there is no clock signal within a microcomputer instruction set. Instead we must assume that events are triggered by execution of an appropriate instruction, rather than a clock signal transition.

How will we represent outputs Q and \bar{Q} ? Two bits of memory could be used to represent these two outputs:

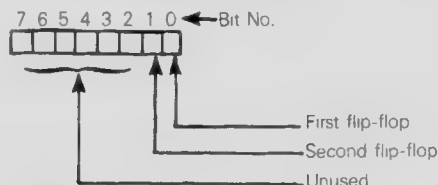


Since we are dealing with data, not signals, \bar{Q} is redundant. The single flip-flop therefore devolves to one memory bit. A 7474 device, since it contains two flip-flops, devolves to two memory bits, one for each flip-flop implemented on the chip.

There is nothing surprising about this conclusion. Each bit of a microcomputer's read/write memory is a simple, bistable element; it could, indeed, be a flip-flop.

The logic of a 7474 flip-flop may be represented by instructions that clear a memory bit, set the memory bit to 1, or store an unknown binary digit in the memory bit.

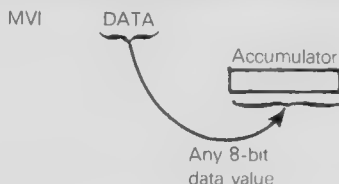
Suppose memory bits are assigned as follows:



The 7474 function table now becomes these instructions:

	Preset	Clear	D	First flip-flop	Second flip-flop
or	L	H	X	MVI 1	MVI 2
	H	H	H	STA FLP	STA FLP
or	H	L	X	MVI 0	MVI 0
	H	H	L	STA FLP	STA FLP
	L	L	X	Does not apply	

With regard to the table above, the MVI instruction acts on the Accumulator contents as follows



The STA instruction stores the resulting Accumulator contents in a memory word identified by the label FLP. Bits 0 and 1 of the memory word identified by FLP are presumed equivalent to the 2 flip-flops of the 7474 device.

MICROCOMPUTER SIMULATION OF FLIP-FLOPS IN GENERAL

In conclusion, a flip-flop becomes a single bit of read-write memory within a microcomputer system.

Within a microcomputer system, all flip-flops are the same. Flip flop logic reduces to these four questions:

- 1) When do I execute an instruction to set a memory bit to 1?
- 2) When do I execute an instruction to reset a memory bit to 0?
- 3) When do I execute an instruction to store a binary digit in a memory bit?
- 4) When do I execute an instruction to read the contents of a memory bit?

THE MICROCOMPUTER SIMULATION OF REAL TIME DEVICES

There are two types of real time devices that we will look at: the one-shot (including monostable multivibrators) and the master-slave flip-flop. Specifically, these devices will be described:

- The Signetics 555 monostable multivibrator
- The 74121 monostable multivibrator
- The 74107 dual J-K master-slave flip-flop with Clear

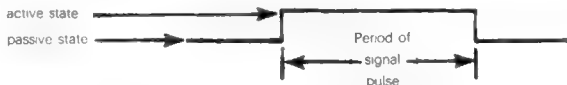
A one-shot is a device which generates a signal pulse with a specific time period:

ONE-SHOT



A monostable multivibrator is a device with one stable, or passive state. It produces one-shot output signals, as illustrated above, where the pulse is in the unstable, or active state:

**MONOSTABLE
MULTIVIBRATOR**



The device is a "multivibrator" because it can output a continuous stream of signals — much like a clock signal. In other words, a multivibrator output consists of a continuous stream of one-shot signals.

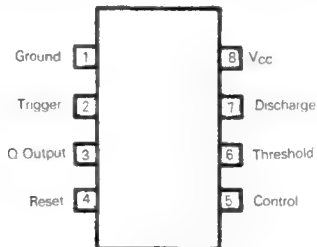
The time period of the signal pulse is a real time value — it is a finite number of microseconds, or milliseconds, or even seconds.

A master-slave flip-flop is a flip-flop which generates output signals based on the condition of input signals at some earlier time. Again we encounter a real time value — the delay between inputs and outputs.

**MASTER-SLAVE
FLIP-FLOP**

THE 555 MONOSTABLE MULTIVIBRATOR

The Signetics 555 monostable multivibrator may be illustrated as follows:



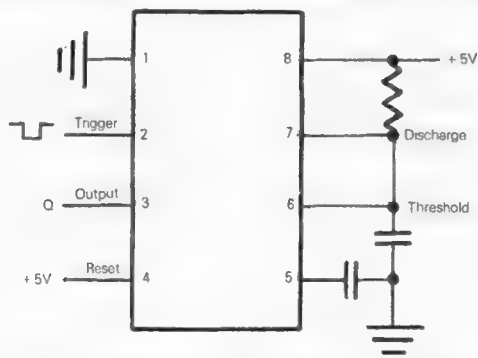
The negative edge of a clock signal at the Trigger input (pin 2) causes a negative-to-positive transition at the Output Q. The duration of the high level output at Q is controlled by a resistor/capacitor circuit connected to the Discharge and Threshold pins (7 and 6, respectively).

Reset is a standard reset input; a low input will hold the Q output low.

The Control pin is used to control voltage within the multivibrator; it is not significant to an overall understanding of how the 555 device works.

The ground and power pins (1 and 8, respectively) are self-explanatory.

Here is one way in which the 555 monostable multivibrator may be configured:



As soon as a high-to-low signal level is sensed at the Trigger input, the capacitor between pin 6 and ground charges. Signal levels at the threshold and discharge pins, as controlled by the resistor R and the capacitor C, control the period for which Q will output high. This time period is given by the following equation:

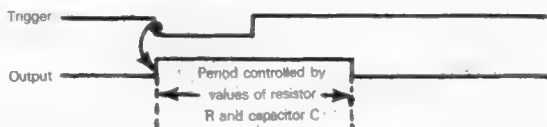
$$T = 1 \cdot 1RC$$

Where T is time in seconds

R is resistance in Megohms

C is capacitance in microfarads

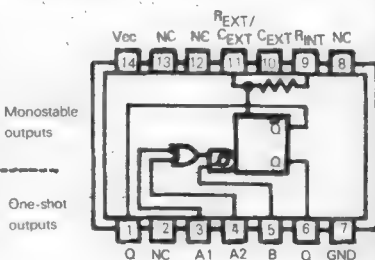
An output signal pulse is generated as follows:



THE 74121 MONOSTABLE MULTIVIBRATOR

The 74121 monostable multivibrator may be illustrated as follows:

FUNCTION TABLE					
INPUTS			OUTPUTS		
A1	A2	B	Q	Q̄	
L	X	H	L	H	Monostable outputs
X	L	H	L	H	
X	X	L	L	H	
H	H	X	L	H	
H	↓	H	↓	↓	One-shot outputs
↓	↓	H	↓	↓	
L	↓	H	↓	↓	
X	L	↓	↓	↓	



A constant low input at A1, A2 or B will hold the 74121 monostable multivibrator in its stable condition with a low Q output and a high \bar{Q} output. High inputs at A1 and A2 have the same effect.

There are five input signal combinations that will generate one-shot outputs. These input signal combinations are identified in the function table above.

With regard to the function table, symbols are used as follows:

X : represents a "don't care"

↓ : represents a one-to-zero logic transition

↑ : represents a zero-to-one transition

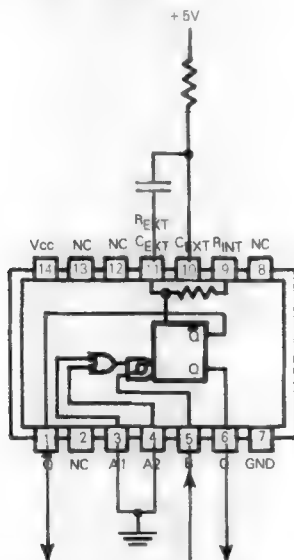
⎓ represents a one-shot with a zero monostable logic level and a one pulse level

⎓ is the NOT of ⎓

The duration of the one-shot output is determined by a resistor-capacitor network, just as described for the Signetics 555 monostable multivibrator, but there are some differences. The 74121 provides an internal resistor which may be accessed by connecting R_{INT} (pin 9) to V_{CC} (pin 14). A variable external resistor may be connected between R_{INT} (pin 9) or R_{EXT} (pin 11) and V_{CC} (pin 14).

An external timing capacitor, if present, will be connected between C_{EXT} (pin 10) and R_{EXT} (pin 11).

Here is one way in which a 74121 monostable multivibrator may be connected:



This use of the 74121 monostable multivibrator corresponds to the bottom two lines of the function table.

An external resistor/capacitor network controls one-shot pulse duration. Each one-shot pulse will be triggered by a low-to-high transition at pin 5 (B).

From the programming point of view, there are only two significant features of the 74121 monostable multivibrator:

- 1) **The monostable outputs are equivalent to binary digits of fixed value.** Any immediate instruction which loads a zero or a one into any register bit simulates the monostable output. Here is an example:





```
MVI    B,4      ;SET BIT 3 OF REGISTER B TO 1. RESET ALL OTHER BITS
```

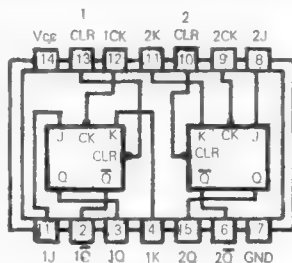
Bit 3 of Register B is equivalent to a flip-flop; so is every other bit of Register B, and every other register.

- 2) **A one-shot output becomes a time delay of fixed value.** We will show how this time delay may be computed within a microcomputer system but first let us examine the 74107 master-slave flip-flop.

THE 74107 DUAL J-K MASTER-SLAVE FLIP-FLOP WITH CLEAR

Consider the 74107 master-slave flip-flop. This flip-flop is illustrated as follows:

FUNCTION TABLE						
INPUTS				OUTPUTS		
1CLR or 2CLR	1CK or 2CK	1J or 2J	1K or 2K	1Q or 2Q	1 \bar{Q} or 2 \bar{Q}	
L	X	X	X	L	H	
H		L	L	Stay as you were		
H		H	L	H	L	
H		L	H	L	H	
H		H	H	Change state regardless of previous state		



 identifies a clock pulse; the way in which it is used is described below.

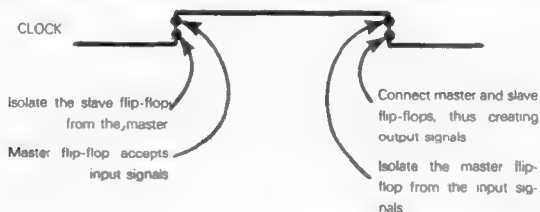
X means "don't care".

Let us examine the function table illustrated above. Unless you are familiar with this type of logic device, its features are not self-evident.

The connotation "master-slave" identifies a circuit which is, in fact, two flip-flops. Therefore, there are four flip-flops in the 74107 device illustrated above.

MASTER-SLAVE FLIP-FLOPS

The flip-flops in each master-slave pair respond to a clock signal as follows:

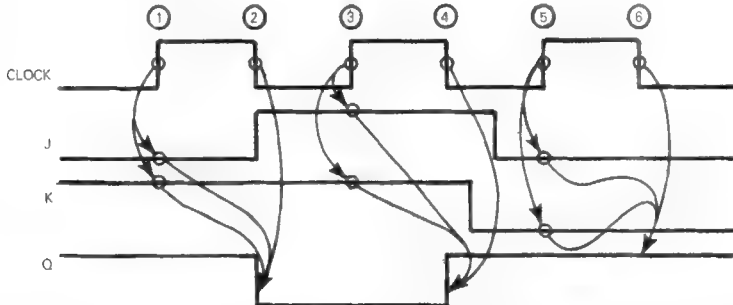


The significance of this clock signal response is that the flip-flop inputs must be present at the positive edge of the clock signal; these inputs must remain steady while the clock signal is high. The flip-flop outputs, however, do not change state until the negative edge of the clock signal.

The Clock signal may be used to create time delays. The 74107 flip-flop output is determined by input signal levels as they existed some time period earlier. This may be illustrated as follows:



Here is a specific example:



The following description of the timing diagram illustrated above is keyed to the circled numbers above the clock signal.

At ②, the Q output goes low, because at ① J was low and K was high.

At ④, Q changes state because at ③ J and K were both high.

At ⑥, Q remains unaltered because at ⑤ J and K were both low.

MICROCOMPUTER SIMULATION OF REAL TIME

What is the significance of the 555 monostable multivibrator and the master-slave flip-flops? When it comes to microcomputer simulation of these devices, there is only one feature that is important to our present discussion — and that is the concept of real time.

The 555 monostable multivibrator creates high logic level pulses at its output, where the duration of the high logic level is a controllable real time function.

The 74107 master-slave flip-flop allows an output signal to be generated based on input conditions as they existed some real time earlier.

MICROCOMPUTER TIMING INSTRUCTION LOOPS

It is simple enough to create a time delay using a microcomputer system — providing the microcomputer system is not being called upon to perform any other simultaneous operations. Consider the following instruction sequence:

**TIMING
SHORT TIME
INTERVALS**

Cycles

	MVI	A, TIME	:LOAD TIME CONSTANT INTO ACCUMULATOR
5	LOOP	DCR A	:DECREMENT ACCUMULATOR
10		JNZ LOOP	:REDECREMENT IF NOT ZERO

The above instruction sequence loads a data value, represented by the label TIME, into the Accumulator. The Accumulator is decremented until it reaches zero, at which time program execution continues. Let us assume that a 500 nanosecond clock is being used by the microcomputer system. The DCR and JNZ instructions, taken together, execute in 15 cycles — which is equivalent to 7.5 microseconds. This means that the program sequence illustrated above can cause a delay with a minimum value of 7.5 microseconds (when TIME equals 1), increasing in 7.5 microsecond steps to a maximum delay of 1920 microseconds, which is equivalent to 7.5×256 . This maximum time delay will result when TIME has an initial value of zero, since TIME is decremented BEFORE being tested to see if it is zero, therefore the time out occurs when 1 decrements to 0, not when 0 decrements to FF_{16} .

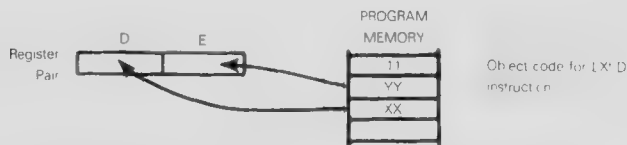
Longer time delays may be generated by having a 16-bit counter. Here is the appropriate instruction sequence:

**TIMING
LONG TIME
INTERVALS**

Cycles

	LXI	D,T16	:LOAD TIME CONSTANT INTO D AND E
5	LOOP	DCX	D :DECREMENT DE
5		MOV	A,D :TEST FOR ZERO BY ORING
4		OR	E :D AND E CONTENTS VIA ACCUMULATOR
10		JNZ	LOOP

The LXI instruction loads a 16-bit value, represented by the label T16, into the DE register pair. The LXI instruction, being an immediate instruction, creates three bytes of object code. When the LXI instruction executes, this is what happens:



The DCX instruction decrements the 16-bit value in the DE registers as a single data entity. However, a quirk of the 8080 instruction set neglects to set status bits based on the result of the 16-bit decrement. This means that we have no immediate way of knowing whether the DE registers now contain a zero, or nonzero value. To make this test, we load the contents of the D register into the Accumulator, then OR with the contents of the E register. If the result in the Accumulator is 0, then both D and E registers must contain 0. If the result is not zero, we return and redecrement the 16-bit value.

**STATUS TESTING
USING DCX
INSTRUCTION**

Observe that 24 cycles are required to travel once through the long time interval instruction loop. Again, assuming that the microcomputer is being driven by a 500 nanosecond clock, it will take 12 microseconds to execute the instruction loop once. The minimum value that T16 may have is 1. The maximum value is again 0, because a decrement occurs before the test for 0; should 0 initially be loaded into D and E, it will be decremented to $FFFF_{16}$ before the first test for zero is made. Thus, the long time interval instruction loop will generate delays that vary in 12 microsecond increments, from a minimum of 12 microseconds to a maximum of 0.786432 seconds.

$$FFFF_{16} = 65535_{10}$$

$$12 \times 65536 = 786432 \text{ microseconds}$$

TIME DELAY INITIATION

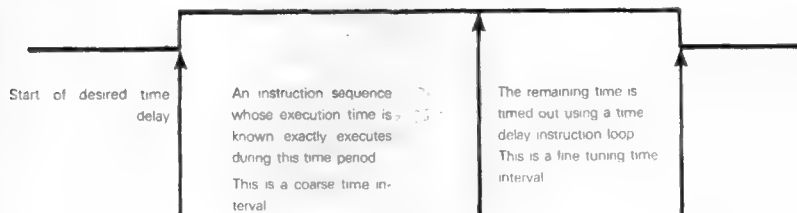


To parallel this concept within a microcomputer program, we must initiate a time delay upon completing some other program sequence's execution. This concept may be illustrated as follows:

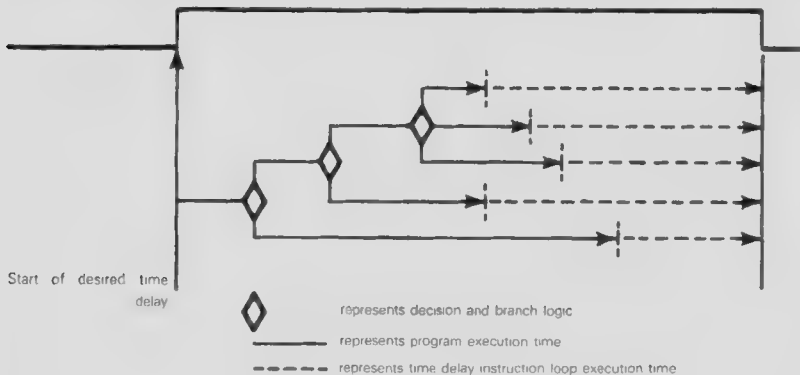
	JMP	DELY	:LAST INSTRUCTION OF SOME PRIOR SEQUENCE
	-		
	-		
	-		
DELY LOOP	MVI DCR JNZ	A,TIME A LOOP	:SHORT TIME INTERVAL INSTRUCTION :SEQUENCE

There is another problem associated with creating time delays within a microcomputer system by executing instruction loops, as we have described: the microcomputer is, in essence, doing no useful work during the time delay. There may be a simple remedy to this problem, providing we can define a program for the microcomputer to execute during the period of the time delay. This follows:

EXECUTING PROGRAMS WITHIN TIME DELAYS



We must assume that we can calculate the exact time it will take for our program to execute within the one-shot time delay; also, the computed time must be less than, or equal to the time delay. Not many programs are going to fit this description. If, for example, more than one instruction sequence may get executed, depending on current conditions, then there may be many different times required for a program to execute. Still, so long as there are a fixed number of identifiable branches, the problem is tractable and may be illustrated as follows:



Now each "limb" of the program branches will end as follows:

```
MVI    A,DLY1    ;LOAD FIRST TIME DELAY
JMP     LOOP      ;START TIME DELAY LOOP
```

-

```
MVI    A,DLY2    ;LOAD SECOND TIME DELAY
JMP     LOOP      ;START TIME DELAY LOOP
```

-

```
MVI    A,DLY3    ;LOAD THIRD TIME DELAY
JMP     LOOP      ;START TIME DELAY LOOP
```

-

```
MVI    A,DLY4    ;LOAD FOURTH TIME DELAY
JMP     LOOP      ;START TIME DELAY LOOP
```

-

```
MVI    A,DLY5    ;LOAD FIFTH TIME DELAY
JMP     LOOP      ;START TIME DELAY LOOP
```

-

-

```
LOOP   DCR     A      ;SHORT TIME INTERVAL INSTRUCTION
        JNZ     LOOP  ;SEQUENCE
```


It is more common than not for a microcomputer program to contain numerous conditional branches; there may be hundreds of different possible execution times depending on various combinations of current conditions. Executing a program within the time interval of the required delay now becomes impractical, because the logic needed to compute remaining time for the innumerable program branches is just too complicated.

THE LIMITS OF DIGITAL LOGIC SIMULATION

An 8080-type microcomputer can compute time delays so long as no other program needs to be executed during the time delay, or providing a very simple instruction sequence with very limited branching is executed during the time delay.

You cannot simulate simultaneous time delays, nor can you simulate a time delay which must occur in parallel to undefinable parallel program executions. External logic must handle all such time delays.

**SIMULTANEOUS
TIME DELAYS**

INTERFACING WITH EXTERNAL ONE-SHOTS

Note that even though external logic may have to create time delays, it is very easy for the microcomputer system to trigger the start of the time delay and for the external logic to report the completion of the time delay.

We can identify the start of a time delay by simply outputting an appropriate binary digit. Look again at the way "Signal Out" was output to external logic by the signal inverter simulation. Outputting a signal to external logic is indeed very easy. Consider the following four instructions:

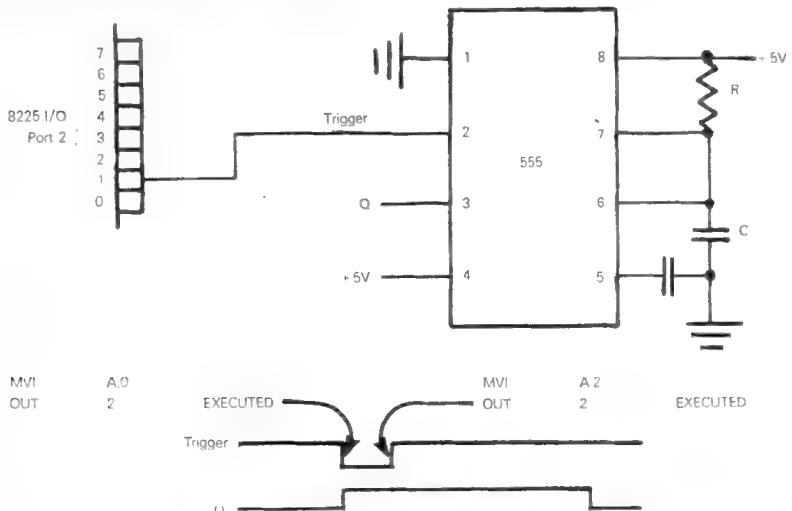
**ONE-SHOT
INITIATION**

```

MVI    A,0      ;LOAD A 0 INTO THE ACCUMULATOR
OUT     2        ;OUTPUT VIA I/O PORT 2
MVI    A,2      ;LOAD A 1 INTO THE ACCUMULATOR BIT 1
OUT     2        ;OUTPUT VIA I/O PORT 2
  
```

A 1 is output at pin 1 of I/O Port 2. Assuming that the pin associated with this I/O port is connected to the trigger of a multivibrator, and that this connection was previously high, then the simple execution of the above instructions will trigger a one-shot.

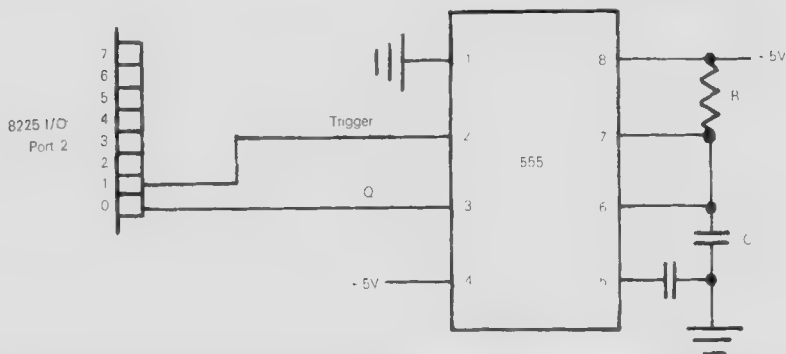
This may be illustrated as follows:



It is equally easy for external logic to signal the end of a time delay.

If we are dealing with "greater than or equal to" logic, all that is necessary is for the one-shot output to be connected to another pin of a microcomputer I/O port:

**ONE-SHOT
TIME OUT
USING
STATUS**



Signals arriving at pins of I/O ports are buffered. The program being executed by the microcomputer may, at any time, input the contents of the I/O port and test the condition of bit 0, which has been wired to the Q output. When this bit is found to equal 0, microcomputer program logic knows that the time interval has been surpassed.

The following instruction sequence will test the I/O port, reset the one-shot input, and clear the "time interval complete" status being reported by I/O Port 2, pin 0:

```
IN      2      ;INPUT CONTENTS OF I/O PORT 2 TO ACCUMULATOR
ANI     1      ;MASK OUT ALL BITS BAR BIT 0
JNZ     NEXT   ;CONTINUE IF BIT IS 1
```

;TIME OUT PROGRAM BEGINS HERE

NEXT ;TIME NOT OUT PROGRAM BEGINS HERE

The IN instruction moves the current contents of I/O Port 2 to the Accumulator.

The following ANI instruction sets all Accumulator bits to 0 bar the bit corresponding to I/O Port 2, pin 0:

7	6	5	4	3	2	1	0	← Bit No.
X	X	X	X	X	X	X	X	Accumulator Contents
0	0	0	0	0	0	0	1	Hexadecimal 01
0	0	0	0	0	0	0	Y	Result of AND

If the binary digit input from pin 0 of I/O Port 2 is 1, then the Q output is still high. The JNZ NEXT instruction simply continues program execution

If bit 0 of I/O Port 2 is 0, then the time delay is over; we branch to a program sequence which only gets executed immediately following a time out.

TIME OUT AND INTERRUPTS

The exact end of a time out can be signaled to the microcomputer using an interrupt.

Now as soon as the one-shot times out, it will force the microcomputer system to cease executing whatever program was currently being executed. A branch will be forced to some other program which has been specifically designed to respond to the time out.

The programming considerations associated with interrupts are more complicated than the level we have been dealing with in Chapter 2. We will therefore defer a detailed description of interrupt processing until later in this book. For the moment it is sufficient to understand that the exact instant of a time out may be signaled to the microcomputer system using interrupt logic.

Chapter 3

A DIRECT DIGITAL LOGIC SIMULATION

The discrete logic devices which we simulated in Chapter 2 were not selected at random; correctly sequenced, they will simulate the logic illustrated in Figure 3-1. This logic is a portion of the printer interface for the Qume Q-Series and Sprint Series printers. Figure 3-2 is the timing diagram that goes with Figure 3-1. We are going to describe both figures at a very elementary level.

Now the purpose of this chapter is to provide a one-for-one correlation between microcomputer assembly language programming and digital logic design. What you must understand is that while such a one-for-one correlation can be forced, it is not natural; and that is where the problem in understanding lies. Microcomputer programs should be written to stress the nature of microcomputers, not the characteristics of digital logic.

The correct way to program a microcomputer is described beginning at Chapter 4.

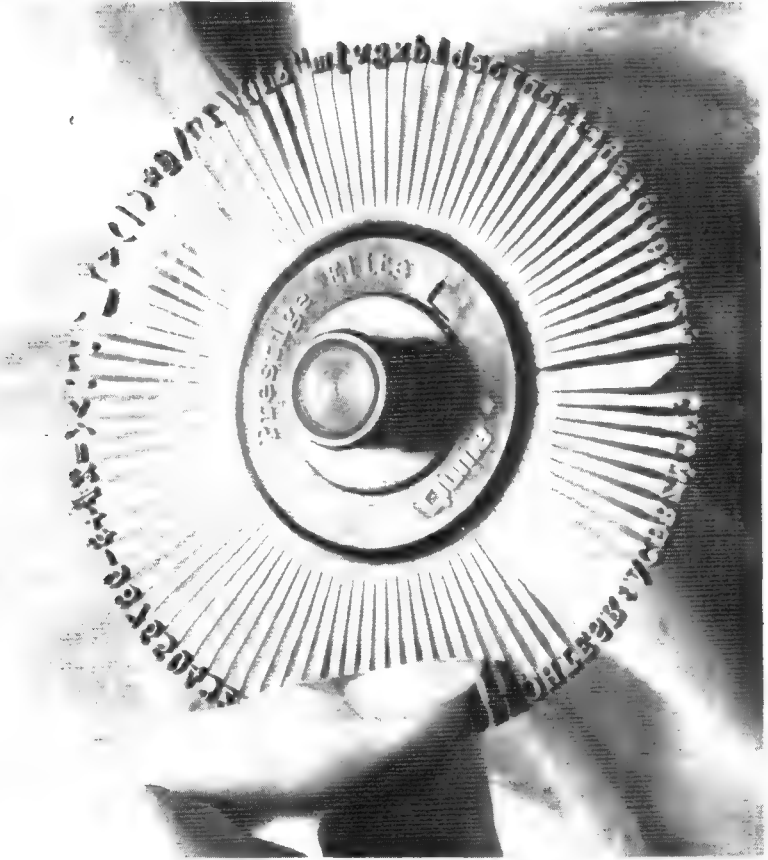
Nevertheless, the juxtaposition of digital logic design and microcomputer programming is underscored in this chapter. This is the chapter that bridges two concepts; and for that reason it is the most important chapter in this book. If you are a logic designer, this chapter is important because it will eliminate digital logic concepts which are inapplicable to microcomputers. If you are a programmer, this chapter is important because it will acquaint you with a new programming goal — efficient logic implementation.

To achieve the goal of this chapter, we will describe the logic illustrated in Figures 3-1 and 3-2; the description will be careful and detailed, so that you can follow this chapter, even if you are not a logic designer. As the logic description proceeds, we will blend in assembly language — in easy stages.

If you understand digital logic, it is particularly important that you confine your reading to the bold face type in this chapter. The logic of Figure 3-1 has been described in sufficient detail to meet the needs of a programmer, or a reader, with no logic background.

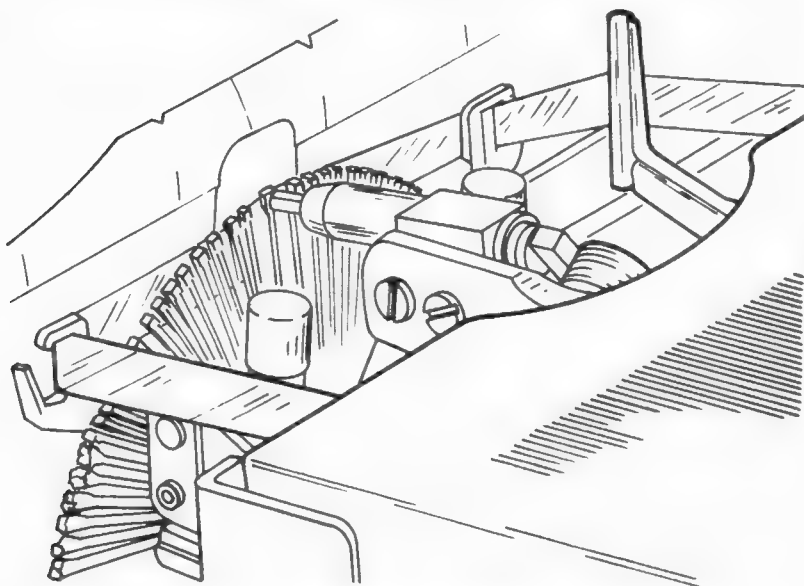
HOW THE QUME PRINTER WORKS

The active Qume printing element is a 96-petal printwheel, with one character on each petal:

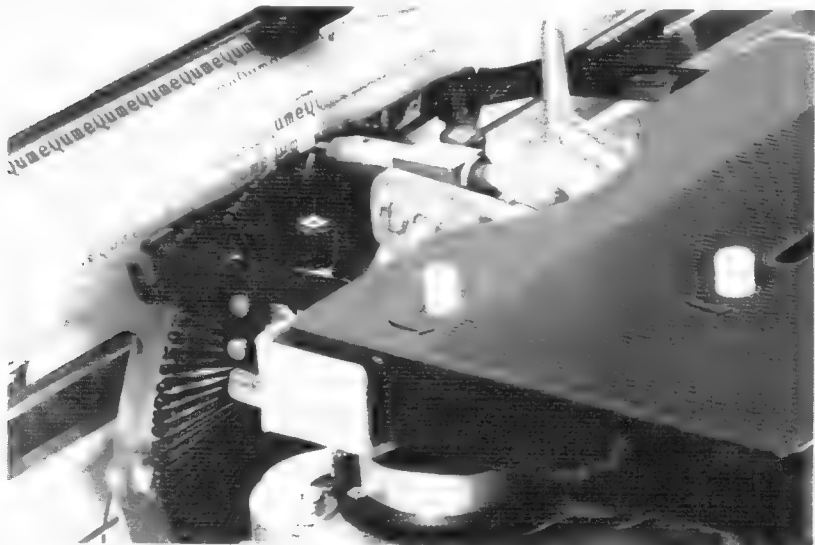


COURTESY OF QUME CORPORATION

A character is printed by moving the printwheel until the appropriate petal is in front of a solenoid driven printhead. The printhead is then fired; it strikes the printwheel petal, which marks the paper:



Whenever a character is not in the process of being printed, the printwheel is positioned with a short petal immediately vertical, so that the character just printed is visible



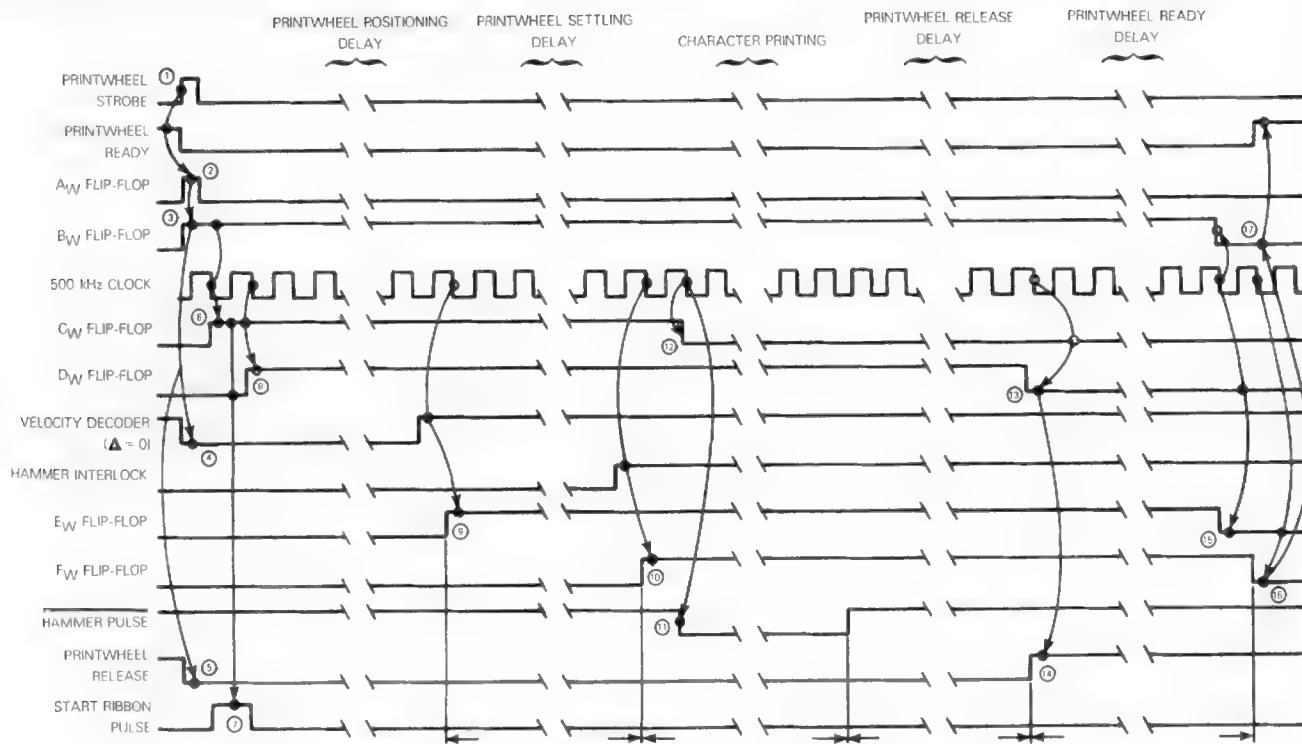


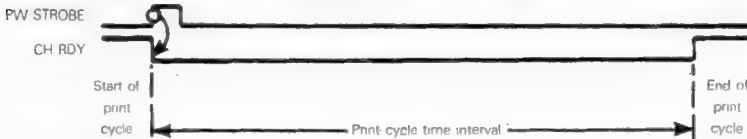
Figure 3-2. Printwheel Control Logic Timing Diagram

Every character is printed according to a definite sequence of events, collectively referred to as a "print cycle". The logic illustrated in Figure 3-1 controls the character print cycle. **These are the events which must occur within a print cycle:**

- PW STROBE



- PRINTWHEEL
READY**



The sequence of events which actually cause a character to be printed can now proceed with the assurance that external logic will not attempt to start printing the next character before the current print cycle has gone to completion.

-
- The diagram shows two signals over time. The top signal, PW STROBE, is a single pulse at the beginning of the print cycle. The bottom signal, CH RDY, remains low until a certain point, then goes high. A vertical dashed line marks the start of the print cycle, and another marks the end. The time between these two points is labeled 'Print cycle time interval'. A label 'Variable Move required petal in front of hammer' points to the time between the start of the print cycle and the rising edge of CH RDY.

A variable time delay is needed by the printwheel positioning logic. Obviously it will take longer to position a petal that is far from the position of visibility than to position to an adjacent petal.

-
- The diagram illustrates the timing of the print cycle. It features two horizontal timelines. The top timeline, labeled 'PW STROBE', shows a single rectangular pulse at the beginning. The bottom timeline, labeled 'CH RDY', shows a signal that transitions from low to high at the 'Start of print cycle' and returns to low at the 'End of print cycle'. The 'Print cycle time interval' is marked with a double-headed arrow along the bottom timeline, spanning from the start to the end of the cycle. This interval is subdivided into two sections: a 'Variable' section labeled 'Move required petal in front of hammer' and a 'Fixed' section labeled 'Printwheel setting time'.

Settling time delays are a very important aspect of the logic supporting any type of mechanical movement. It is easy to draw a clean line showing movement velocity as follows:

SETTLING DELAYS



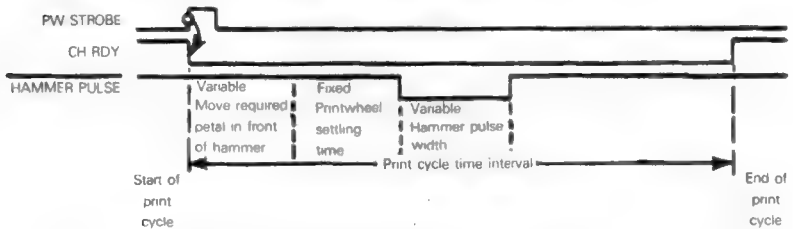
But in reality, movement occurs like this:



The bounce that follows deceleration must be passed over by a settling time delay.

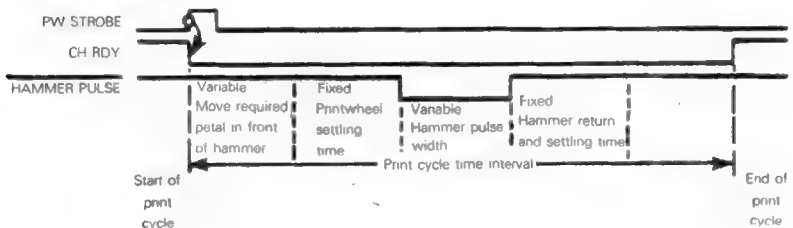
A blurred character will be printed if the printhead is still vibrating when the printhead hits a petal against the paper.

- 5) At the end of the printhead settling time delay, **the printhead can be fired.** This is done by outputting an impulse to a solenoid. **Six firing impulse intensities are provided**, since some characters have a more substantial surface area than others. To strike a comparatively large surface area like a "W" with the same intensity that you strike a small character, like a "i", would produce unevenness in the density of the printed text. The duration of the printhead solenoid pulse is controlled by the next time delay:



The bar over HAMMER PULSE identifies the signal as one which is low when active

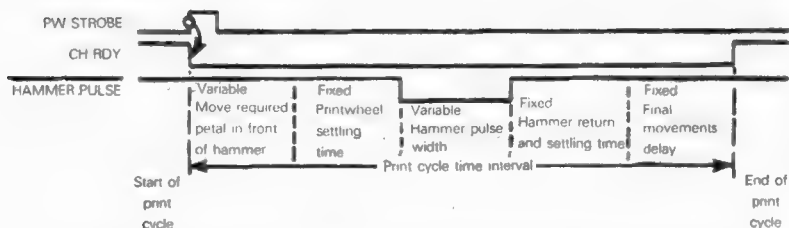
- 6) At the completion of the printhead pulse time delay, the hammer has struck a petal and forced it onto the paper. Now **the hammer must be given time to return to its pre-firing position.** A 3 millisecond delay is generated for this purpose:



- 7) Now **the printwheel can be moved to its position of visibility** and the paper carriage can be advanced to the next character position. The printwheel's "position of visibility" is its normal inactive position; in this position a short petal is in front of the printhead, so the most recently printed character is visible above the short petal; hence the "position of visibility". Had we not given time for the printhead to settle back before moving the printwheel to its position of visibility, a printwheel petal may have been broken, striking the tip of the still protruding hammer. Also the paper may have smudged moving against a bent petal. Since the printhead has been given time to fully retract, none of these problems will arise.

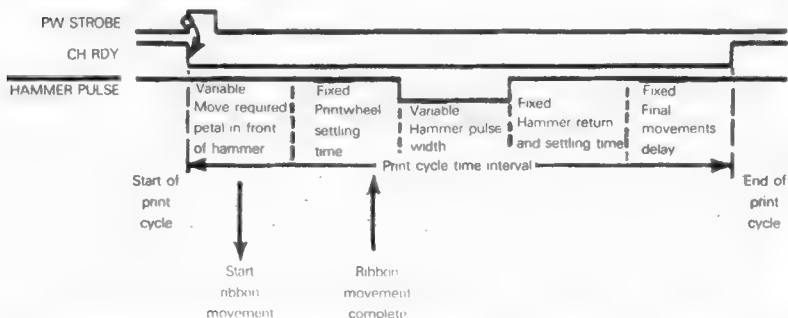
**PRINTWHEEL
POSITION OF
VISIBILITY**

A final 2 millisecond time delay allows the printwheel and paper carriage to reposition themselves:



- 8) What about ribbon logic? **In order to get a clean impression on the paper, a fresh piece of ribbon must present itself between the character petal and the paper.** Shortly after the beginning of the print cycle, therefore, a signal (START RIBBON MOTION PULSE) is output to external logic, which actually controls ribbon movement. This external logic (it is not part of Figure 3-1) sends back a ribbon movement completed signal (FFA) since we cannot allow the printhead to be fired while the ribbon is still moving. Thus **the ribbon is advanced while the printwheel is initially being positioned and settled:**

**START
RIBBON
PULSE**
FFA



In summary, a print cycle consists of 5 time delays, each time delay starts out with a flurry of logical activity, followed by a period of mechanical movement.

INPUT AND OUTPUT SIGNALS

Now that you have a general understanding of the functions which are controlled by logic in Figure 3-1, **the next step is to take a closer look at input and output signals.**

In order to know what to do, and when to do it, we must rely entirely upon input signals. Similarly, output signals represent the only way in which we can transmit control to external logic.

Our limited goal, at this point, is to understand what function each input and output signal performs, and how — physically — we are going to handle the signals. We will discuss the "how" first.

INPUT/OUTPUT DEVICES

There are two types of devices used to transmit signals and data between an 8080 microcomputer system and external logic.

**PROGRAMMABLE
PERIPHERAL
INTERFACE**

First, there are programmable peripheral interface devices, such as the 8255 family of devices manufactured by a number of companies, or the TMS5501 manufactured by Texas Instruments.

Then there are simple latched buffers, such as the 8212.

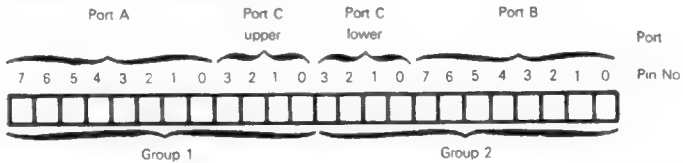
**LATCHED
BUFFER**

We are going to use an 8255 Programmable Peripheral Interface and an 8212 latched buffer.

Since these devices have been described in "An Introduction To Microcomputers", we are going to assume that you understand their capabilities and organization superficially, if you do not, see An Introduction To Microcomputers — Volume II — Some Real Products before continuing, otherwise you will not understand the discussion which follows.

THE 8255 PROGRAMMABLE PERIPHERAL INTERFACE

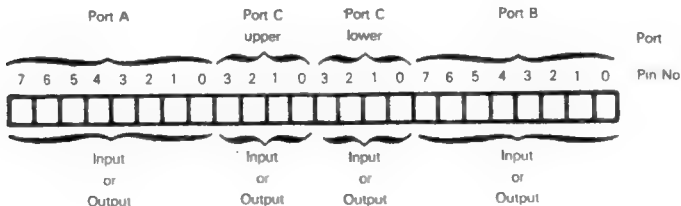
The 8255 Programmable Peripheral Interface provides 24 I/O pins, which may be grouped into ports as follows:



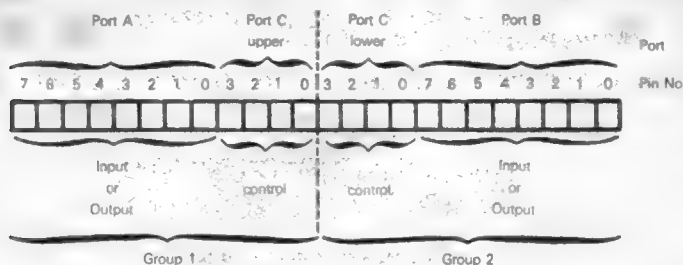
Each group of pins may be programmed to operate in one of three modes. Group 1 and Group 2 ports do not have to operate in the same mode.

**I/O PORT
MODES**

In Mode 0, every port is either a simple input port, or a simple output port:

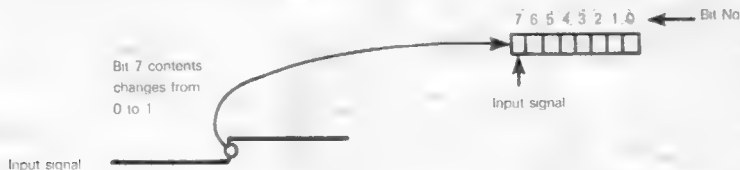


In Mode 1, the A (or B) Port is either a strobed input port, or a strobed output port. Strobe and control signals are provided by the C Port pins in the same group.



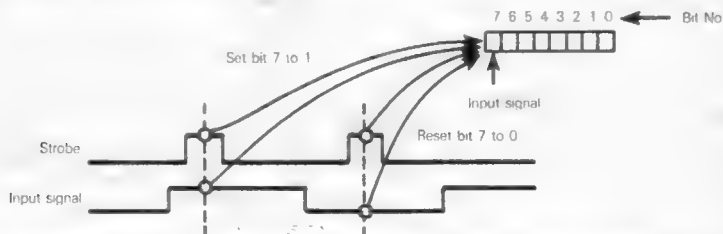
What is the difference between a basic (Mode 0) I/O port and a strobed (Mode 1) I/O port? The basic I/O mode accepts and transmits data immediately. An input signal changing state will immediately set or reset an appropriate bit within the I/O port buffer:

SIMPLE I/O

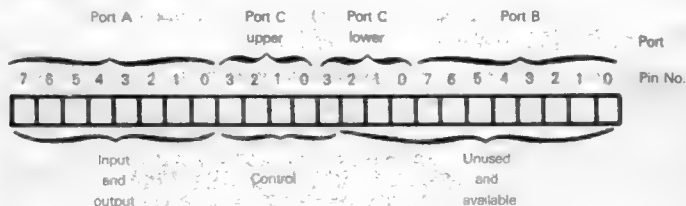


A strobed I/O port uses control signals to determine the instant at which a data change will be recorded. Again, with reference to an input signal, this may be illustrated as follows

STROBED I/O



8255 I/O pins may also be programmed to operate as a single, bidirectional 8-bit I/O port supported by five control signals:



As illustrated above, the 8255 PPI is being used in Mode 2.

Irrespective of mode, every PPI has four I/O port addresses assigned to it. Three 8255 pins are used to select the device, and a device port, as follows

I/O PORT ADDRESSING

CS: Input 0 to select the device. Input 1 to disconnect it

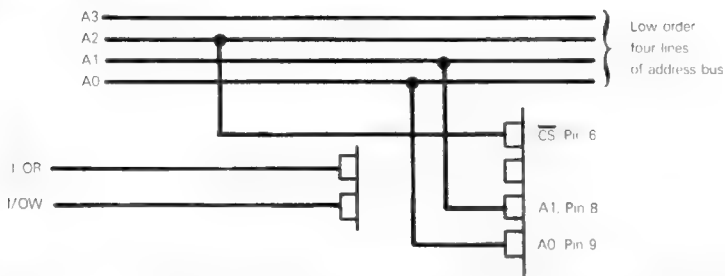
A1	A0	
0	0	Address Port A
0	1	Address Port B
1	0	Address Port C (upper and lower, as a single 8-bit unit)
1	1	Address PPI device's "control port"

The device "control port" is a write only port. You select port modes by writing an appropriate code into the control port. A detailed discussion of control port codes will not help you understand the subject matter of this chapter, so we leave that discussion to An Introduction To Microcomputers — Volume II — Some Real Products.

I/O PORT MODE SELECTION

Now when an IN or OUT instruction is executed by an 8080-type CPU, the port number is output on the low order eight address bus lines. Therefore, we will connect our 8255 PPI as follows:

I/O PORT ADDRESS DETERMINATION



These pin connections will cause 8255 I/O ports to be addressed as follows:

	A2	A1	A0
Port 0 - PPI			
Port A	0	0	0
Port 1 - PPI			
Port B	0	0	1
Port 2 - PPI			
Port C	0	1	0
Port 3 - PPI			
Control Port	0	1	1

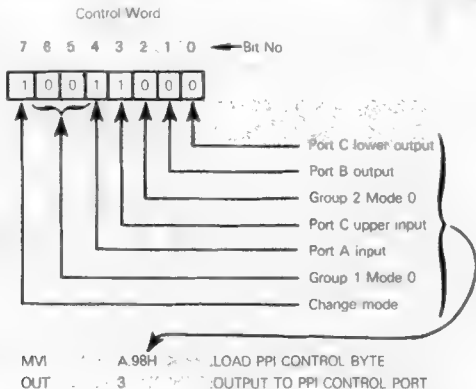
Notice that you must decode the address bus with care when using simple I/O devices such as the 8212 I/O port. The address bus of 8080-type microprocessor devices outputs diagnostic information when the address bus is supposedly inactive. By ANDing the chip select with appropriate Bus Control signals, in this case I/O OR1/O W, we insure that the address bus is decoded only when an I/O operation is being initiated.

ADDRESS BUS DECODE

Initially, to keep things simple, we are going to program the 8255 PPI to operate in Mode 0, with Group 1 ports assigned to input and Group 2 ports assigned to output.

In order to understand the discussion at hand, you do not need to know how the 8255 PPI is programmed to meet our requirements; nevertheless here is the appropriate instruction sequence and control word explanation:

I/O PORT MODE SELECT INSTRUCTION SEQUENCE

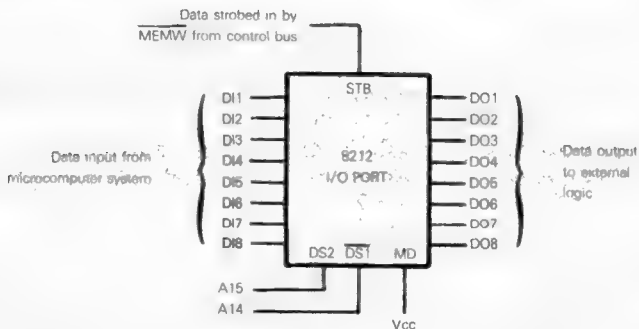


To verify the above control word format, compare it with the description of the 8255 PPI in An Introduction To Microcomputers—Volume II—Some Real Products.

THE 8212 8-BIT INPUT/OUTPUT PORT

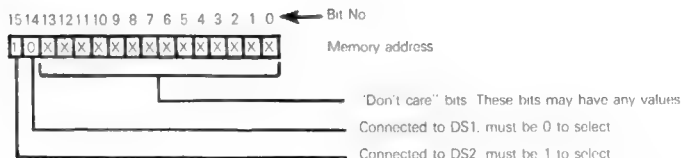
This device is simply a strobed 8-bit input/output buffer.

The 8212 I/O port has eight "data in" pins and eight "data out" pins. I/O port mode is controlled by signal MD and data transfers are strobed by signal STB. We are going to use the 8212 I/O port as a data storage and output device, wired as follows:



We have wired the 8212 I/O port to select itself in response to memory reference instructions, providing the two high order bits of the memory address are 10. In other words, any memory address in the range 8000H through BFFF will select the 8212 I/O port:

**I/O PORTS
ADDRESSED
AS MEMORY**



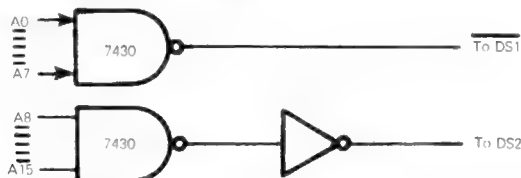
If you look at the range of values the "don't care" bits can have you get the range of memory addresses to which the 8212, as wired, will respond:

Minimum address: 1000000000000000
 8 0 0 0

Maximum address: 1011111111111111
 B F F F

We have used up $3FFF_{16}$ memory addresses accessing a single I/O port. Does that matter? Not really. We still have $C000_{16}$ addresses left — far more than we will need

We could have reserved just one memory address for our I/O port, but that would have meant synthesizing the two select signals out of 16 address lines. Here is how the single address $FFFF_{16}$ could select the 8212 I/O port:



A 7430 is an 8-input NAND gate.

Remember that the 7430 outputs must be ANDed with appropriate control signals to insure that the address bus is decoded only when a valid address is being output.

But why waste money on the unnecessary extra logic?

INPUT SIGNALS

Let us now turn our attention to the input signals that appear on the left-hand side of Figure 3-1. We will describe each signal, assign it to an appropriate input pin, and include a rudimentary instruction sequence to access the signal at the most elementary level.

RETURN STROBE

If the operator is to see the most recently printed character, two things must happen:

- 1) The printwheel must be moved to its position of visibility.
- 2) The ribbon must be dropped.

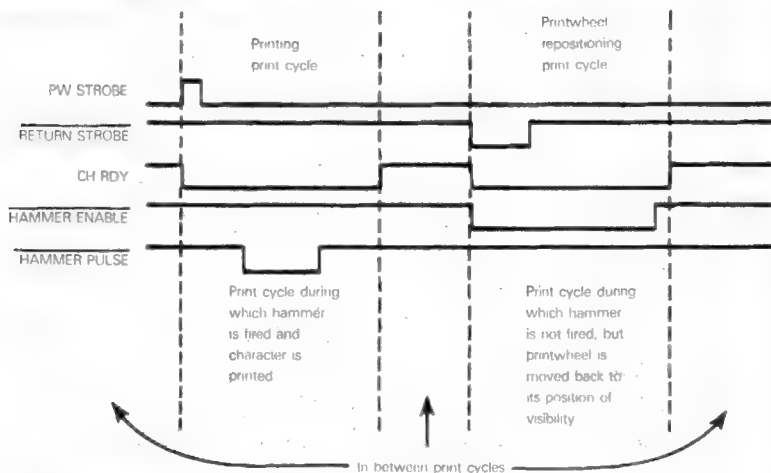
External logic can take care of dropping and raising the ribbon, but logic in Figure 3-1 creates the signals that allow the printwheel to move.

In order to move the printwheel to its position of visibility, therefore, the ribbon control external logic inputs RETURN STROBE low while the ribbon is dropped.

Logic within Figure 3-1 uses RETURN STROBE as an alternative signal to start a print cycle; however, RETURN STROBE low is accompanied by HAMMER ENABLE FF low, which prevents the printhead from firing. Therefore, a print cycle initiated by RETURN STROBE

**PRINTWHEEL
REPOSITIONING
PRINT CYCLE**

low is a "dummy" print cycle which moves the printwheel back to its position of visibility, but does not fire the printhead; we refer to this as a printwheel repositioning print cycle:



We will assign I/O Port 2, pin 4 to RETURN STROBE.

In between print cycles, we can test this pin in order to trigger a new print cycle via the following instruction sequence:

```

LOOP    IN      2      ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        ANL     10H    ;MASK OUT ALL BAR BIT 4
        JNZ     LOOP   ;IF THIS BIT IS 1, RETURN AND RETEST
;NEW PRINT CYCLE INSTRUCTION SEQUENCE BEGINS HERE
    
```

Here is a check on how the ANI instruction works in the sequence above:

7	6	5	4	3	2	1	0	← Bit No.
X	X	X	Y	X	X	X	X	, Accumulator contents
0	0	0	1	0	0	0	0	10H
0	0	0	Y	0	0	0	0	AND

↑ This bit corresponds to RETURN STROBE

PFL REL

The printhammer cannot be fired while the paper feed mechanism is moving, therefore at such times external logic inputs PFL REL low.

Logic within Figure 3-1 will delay firing the printhammer for as long as PFL REL is being input low.

We will assign Pin 0 of I/O Port 0 to PFL REL.

Before executing the instruction sequence which fires the printhammer, we will input the contents of Port 0 and test bit 0, so long as this bit contains zero, we will not execute the printhammer firing sequence.

The following instructions perform the required test:

```
LOOP    IN      0      ;INPUT CONTENTS OF I/O PORT 0 TO ACCUMULATOR
        ANI     1      ;MASK OUT ALL BITS BAR BIT 0
        JZ      LOOP    ;THE BIT IS 0 DO NOT FIRE THE PRINTHAMMER
;PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE
```

RIB LIFT RDY

This signal is similar to PFL REL, it **is input low when ribbon lift logic is moving the ribbon**. Just as the printhammer cannot be fired while the paper feed mechanism is active, so it cannot be fired while the ribbon is being moved. By connecting RIB LIFT RDY to Pin 1 of I/O Port 0, we may adjust the printhammer firing initiation instruction sequence as follows

```
LOOP    IN      0      ;INPUT CONTENTS OF I/O PORT 0 TO ACCUMULATOR
        ANI     3      ;MASK OUT ALL BITS BAR 0 AND 1
        CMA     ;COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT PRE-
                SENT
        JNZ     LOOP    ;ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS NOW 1, DO NOT
                FIRE PRINTHAMMER
;PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE
```

PW STROBE

We have already encountered **this signal it is pulsed high by external logic to start a normal print cycle**, during which a character will be printed.

Remember, RETURN STROBE is input low to initiate a print cycle during which the printwheel will be moved to its position of visibility, but no character will be printed.

Assuming that **PW STROBE is connected to pin 5 of I/O Port 2**, this is the instruction sequence that will be executed between print cycles:

```
LOOP    IN      2      ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        ANI     30H     ;ISOLATE BITS 5.(PW STROBE) AND 4.(RETURN STROBE)
        CPI     10H     ;TEST FOR PW STROBE = 0, RETURN STROBE = 1
        JZ      LOOP    ;IF TEST IS TRUE, STAY IN LOOP
;PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE
```

Observe that either $PW\ STROBE = 1$, or $RETURN\ STROBE = 0$ must trigger the start of a print cycle; that is why only $PW\ STROBE = 0$ and $RETURN\ STROBE = 1$ keeps us in the testing instruction loop.

Now the four instructions shown above execute in a combined total of 34 clock cycles. With a 500 nanosecond clock, the four instructions will execute in 17 microseconds — which becomes the minimum pulse width allowed for PW STROBE. **If PW STROBE is pulsed high for less than 17 microseconds, our instruction cycle may miss it.**

**INPUT SIGNAL
PULSE WIDTH**

FFA

This is another printhammer warning signal. It is set to 0 while external logic is advancing the ribbon. By connecting this signal to pin 2 of I/O Port 0, we can modify the instruction sequence which precedes printhammer firing as follows:

```
LOOP    IN      0      ;INPUT CONTENTS OF I/O PORT 0 TO ACCUMULATOR
        ANI     7      ;ISOLATE BITS 2, 1 AND 0
        CMA     ;COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT
        JNZ     LOOP   ;ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS 1, DO NOT FIRE
                        PRINTHAMMER
```

;PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE

All we have done is add one more test condition which must be met before the printhammer firing instruction sequence gets executed.

RESET

This is a signal which is commonly seen in the most diverse types of logic. It is an initializing signal. Its purpose is to ensure that all logic is in a "beginning" state, which in our case is the condition which exists between printwheel cycles.

The logic in Figure 3-1 connects the RESET signal to logic devices such that RESET going high forces all logic to a "beginning" condition.

There are many ways in which a microcomputer system can handle a RESET signal. The simplest scheme is to input this signal to the RESET pin of the 8080 CPU.

RESET
THE CPU

Another method of handling RESET is to test the signal in between print cycles and to prevent any print cycle from starting while RESET is high; this may be accomplished by connecting RESET to pin 6 of I/O Port 2, then modifying our "in between print cycles" instruction sequence as follows:

```
LOOP    IN      2      ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
        ANI     40H     ;ISOLATE BIT 6 (RESET)
        JNZ     LOOP   ;IF RESET IS HIGH, STAY IN LOOP

;RESET IS LOW. TO TEST PW STROBE AND RETURN STROBE
IN      2      ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
ANI     30H     ;ISOLATE BIT 5 (PW STROBE) AND BIT 4 (RETURN STROBE)
CPI     10H     ;TEST FOR PW STROBE = 0, RETURN STROBE = 1
JZ      LOOP   ;IF TEST IS TRUE STAY IN LOOP
```

;PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE

Now this longer test loop will require 61 cycles to execute. That means PW STROBE must pulse high for at least 30.5 microseconds, assuming a 500 nanosecond clock.

SIGNAL
PULSE
WIDTH

PFR REL

This is yet another signal which must be tested before initiating printhammer firing. It indicates when external logic is moving the paper feed. Under such circumstances we cannot fire the printhammer. By connecting this signal to pin 3 of input Port 0, we merely have to adjust the printhammer firing instruction initiation sequence as follows:

```
LOOP    IN      0      ;INPUT CONTENTS OF I/O PORT 0 TO ACCUMULATOR
        ANI     0FH     ;ISOLATE BITS 3, 2, 1 AND 0
        CMA     ;COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT
        JNZ     LOOP   ;ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS 1, DO NOT FIRE
                        PRINTHAMMER
```

;PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE

CA REL

This signal is almost identical to PFR REL. It **comes from external logic that controls carriage movement. We will connect this signal to pin 4 of input Port 0** and modify the hammer firing instruction initiation sequence as follows:

```
LOOP  IN      0      ;INPUT CONTENTS OF I/O PORT 0 TO ACCUMULATOR
      ANI      1FH    ;ISOLATE BITS 4, 3, 2, 1 AND 0
      CMA      ;COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT
      JNZ      LOOP   ;ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS 1, DO NOT FIRE
                        PRINTHAMMER
```

;PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE

FFI

This is the signal which times the first delay in the print cycle — the time during which the printwheel moves from its position of visibility until the required petal is in front of the printhead.

FFI is generated by external logic. It is low while the printwheel is moving and it is high while the printwheel is not moving.

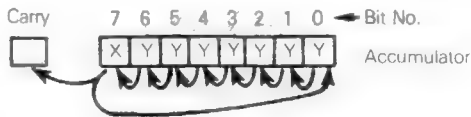
We will tie FFI to pin 7 of I/O Port 0. The following instruction loop will create a delay which lasts until FFI goes high:

```
LOOP  IN      0      ;INPUT PORT 0 TO ACCUMULATOR
      RLC      ;SHIFT BIT 7 INTO THE CARRY
      JNC      LOOP   ;IF CARRY = 0 STAY IN THE LOOP
```

**TIME DELAY
BASED ON
INPUT SIGNAL**

Do you see how this loop works? After I/O Port 0 contents have been input to the Accumulator, we are only interested in bit 7, since this is the bit that corresponds to FFI.

This is what the RLC instruction does:



If the Carry status equals 1, the printwheel move delay is over. If Carry equals 0, program logic must continue the delay.

EOR DET

This signal indicates that the end of the ribbon has been reached. Under these circumstances character printing cannot continue.

When this signal is generated, there will still be fresh ribbon in front of the printhead, so the signal is not used to inhibit printhead firing. Rather it is used to prevent the end of the print cycle from ever being indicated. This effectively prevents a new print cycle from ever starting.

We will connect the EOR DET signal to bit 7 of I/O Port 2. Since EOR DET is a negative logic signal we will test it prior to going into the "in between print cycle" loop as follows:

```

:TEST FOR VALID END OF PRINT CYCLE
LOP1  IN      2      ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
      RLC          ;SHIFT BIT 7 INTO CARRY
      JNC      LOP1  ;IF ZERO IN CARRY, STAY IN PRINT CYCLE
:START OF IN BETWEEN PRINT CYCLES LOOP
LOOP  IN      2      ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
      ANI      40H   ;ISOLATE BIT 6 (RESET)
      JNZ      LOOP  ;IF RESET IS HIGH, STAY IN LOOP
:RESET IS LOW. TO TEST PW STROBE AND RETURN STROBE
      IN      2      ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
      ANI      30H   ;ISOLATE BITS 5 (PW STROBE) AND 4 (RETURN STROBE)
      CPI      10H   ;TEST FOR PW STROBE = 0, RETURN STROBE = 1
      JZ       LOOP  ;IF TEST IS TRUE STAY IN LOOP

```

:PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE

Look at the above instruction sequence. There are some interesting aspects to it.

The first three instructions above will be the last three instructions in the print cycle sequence. The instruction labeled LOOP is the first instruction of a sequence which gets executed continuously until the start of the next print cycle. Thus, if EOR DET is low, program logic will hang up in the first three instructions listed above, constantly looping within these three instructions until EOR DET goes high. At that time, the print cycle ends and we go into the "in between print cycles" instruction loop. The program now hangs up indefinitely in this instruction loop until bit 6 which corresponds to RESET equals 0, while bit 5 which corresponds to PW STROBE equals 1, or bit 4 which corresponds to RETURN STROBE equals 0.

There is another interesting feature of the above instruction sequence. We could, if we wished, eliminate the second IN instruction, as follows:

```

:TEST FOR VALID END OF PRINT CYCLE
LOP1  IN      2      ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
      RLC          ;SHIFT BIT 7 INTO CARRY
      JNC      LOP1  ;IF ZERO IN CARRY, STAY IN PRINT CYCLE
:START OF IN BETWEEN PRINT CYCLES LOOP
      ANI      80H   ;ISOLATE BIT 6 (RESET)
      JNZ      LOP1  ;IF RESET IS HIGH, STAY IN LOOP
:RESET IS LOW. TO TEST PW STROBE AND RETURN STROBE
      IN      2      ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
      ANI      30H   ;ISOLATE BITS 5 (PW STROBE) AND 4 (RETURN STROBE)
      CPI      10H   ;TEST FOR PW STROBE = 0, RETURN STROBE = 1
      JZ       LOP1  ;IF TEST IS TRUE STAY IN LOOP

```

:PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE

By eliminating one instruction, we have saved two bytes of object code. The penalty is that we have added 14 clock cycles to the entire instruction loop, which means that the PW STROBE high pulse goes up from the 30.5 microseconds we calculated when discussing the RESET signal to 37.5 microseconds.

Why does the condensed instruction sequence illustrated above work? The reason is because external logic is not supposed to be moving the ribbon in between print cycles, therefore EOR DET will always be high during the in between print cycle instruction execution loop. If this is so, the RLC instruction will always shift a 1 into the Carry, which will always cause execution to continue with the ANI instruction. Thus the first three instructions become harmless. Notice that the ANI 40 instruction has become an ANI 80 instruction since the RESET signal bit has been shifted one position to the left by the RLC instruction.

HAMMER ENABLE FF

This is the signal which prevents the printhammer from being fired after the print-wheel is moved to its position of visibility, as described in connection with the RETURN STROBE signal.

We will connect **HAMMER ENABLE FF** to pin 6 of I/O Port 0, then modify the instruction sequence which precedes printhammer firing as follows:

```
LOOP    IN      0      ;INPUT CONTENTS OF I/O PORT 0 TO ACCUMULATOR
        ANI     5FH     ;ISOLATE BITS 6, 4, 3, 2, 1 AND 0
        CMA     ;COMPLEMENT THE RESULT TO TEST FOR ANY 0 BIT
        JNZ     LOOP    ;ANY 0 BIT WILL NOW BE 1. IF ANY BIT IS 1, DO NOT FIRE
                        PRINTHAMMER
```

;PRINTHAMMER FIRING INSTRUCTION SEQUENCE BEGINS HERE

CLK

This is the clock signal that synchronizes all logic in Figure 3-1. Try as we may, we cannot include this signal in our simulation of Figure 3-1, since events within the microcomputer program are going to be synchronized by the sequence in which instructions are executed — not by a clock. Similarly, the next two signals, +5V and RV1, are power supplies. They are meaningless within a microcomputer program.

H1 - H6

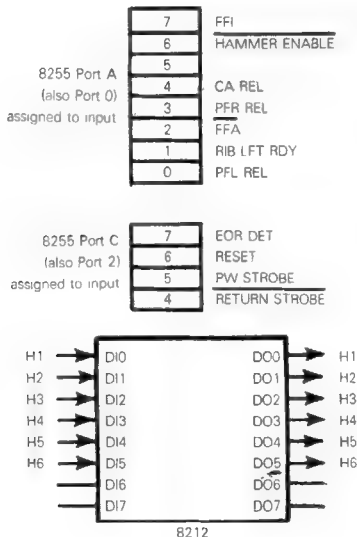
These are the six signals which select one of six time durations for the printhammer firing pulse. We will assign these signals to the 8212 I/O port. Once the printhammer firing instruction sequence gets executed, it simply loads these signals into the Accumulator as follows:

```
LDA      H1H6      ;INPUT FIRING PULSE TIME CODE
```

H1H6 is the four-character label representing the I/O port's "select" memory address.

INPUT SIGNAL SUMMARY

In summary, this is how input signals have been assigned:



OUTPUT SIGNALS

We will now turn our attention to the output signals listed on the right-hand side of Figure 3-1. These signals are much easier to describe than the input signals. They consist of six flip-flop outputs — which are simply timing indicators used by external logic — plus four control signals. We are going to output these signals to the B and C lower ports of the 8255 PPI as follows:

8255 Port B (also Port 1) assigned to output	7	
	6	FFF
	5	FFE
	4	FFE
	3	FFD
	2	FFC
	1	FFB
	0	FFA

8255 Port C (also Port 2) assigned to output	3	START RIB MOTION
	2	HAMMER PULSE
	1	CH RDY
	0	PW RELEASE

We assign a pin for FFC even though it is not output, because I/O Port 1 is going to serve a double purpose — as a data storage location and as an output signals' buffer. Simple routines to generate output signals cannot be concocted; that is the whole purpose of the logic in Figure 3-1. We will therefore simply define the four control signals:

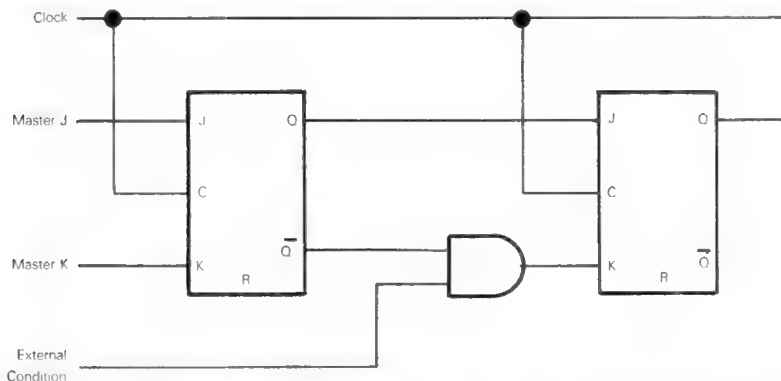
- 1) **PW REL.** This signal marks the end of the fixed printhead return and settling time delay and the beginning of the fixed Final Movement's delay during which external logic can move the paper feed and carriage.
- 2) **CH RDY.** This is also referred to as the PRINTWHEEL READY signal. This is the signal which defines the entire print cycle time interval; it goes low at the start of the print cycle and stays low until the end of the print cycle.
- 3) **HAMMER PULSE.** This signal must be output low for the time interval during which external logic is supposed to transmit a firing pulse to the printhead solenoid.
- 4) **START RIBBON MOTION PULSE.** This signal is pulsed high early in the print cycle, telling external logic that it is safe to begin advancing the ribbon so that fresh ribbon will be in front of the printhead when it is fired.

A DIGITAL LOGIC ORIENTED SIMULATION

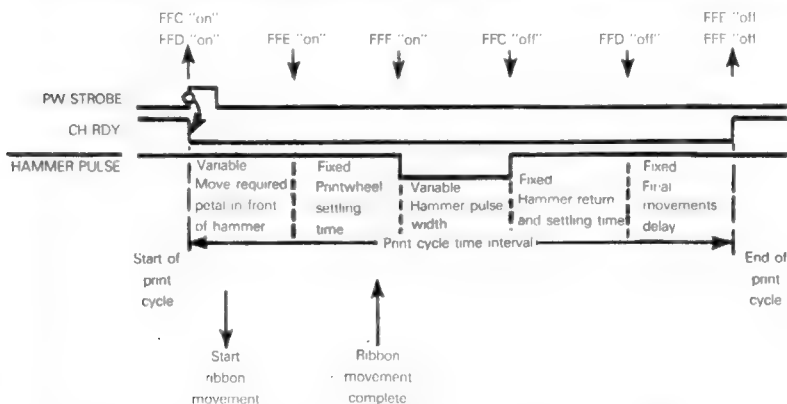
We are now ready to start simulating the logic illustrated in Figure 3-1 — but first a brief overview of the logic.

A LOGIC OVERVIEW

At the center of the logic sequence are four 74107 flip-flops, labeled FFC_W , FFD_W , FFE_W and FFF_W . You will find these flip-flops in the center and to the left of Figure 3-1. These four flip-flops form what is known as a "Johnson Counter". Each flip-flop is controlled by the output of the previous flip-flop, coupled with a test for external conditions:



Thus the four flip flops may be visualized as initiating print cycle events in the following way:



As illustrated above, the print cycle time interval may be divided into five periods

During the first time interval the printwheel is moved from its position of visibility until the required petal is in front of the printhead. This time interval is controlled by external logic, via the FFI input.

The remaining four time intervals are controlled by three 74121 one-shots and the 555 multivibrator.

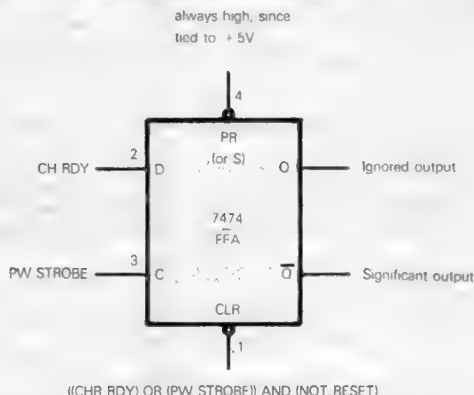
What about the two 7474 flip-flops at the top left hand corner of Figure 3-1? These are simply cycle initiation logic. Flip-flop FFA is triggered by a combination of signals necessary for a print cycle to begin. Flip-flop FFB acts as a switch for the four 74107 flip flops, forcing them to turn 'off' in between print cycles. Flip-flop FFB does this by tying its Q output to the reset inputs of the 74107 flip-flops. This results in the 74107 flip-flops always being turned off if FFB is turned off; we will explain in more detail how this happens later on

We are now going to follow a print cycle through Figure 3-1. As we progress, we will create a microcomputer assembly language program that simulates the logic, device-by-device.

FLIP-FLOP FFA_W

Our print cycle begins at the 7474 flip-flop designated FFA_W. You will find this flip-flop at the top, left hand corner of Figure 3-1. Let us isolate FFA_W, and illustrate it as follows:

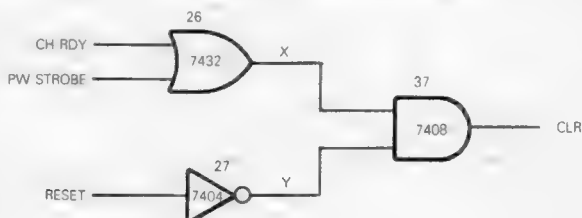
**7474
FLIP-FLOP**



Refer back to the general function table for a 7474 flip-flop, given in Chapter 2.

Since PRESET (PR) is always high, being tied to +5V, a low CLEAR (CLR) input will force the flip flop "off", at which time Q is output low and \bar{Q} is output high.

Look at Figure 3-1 and you will see that CLR is generated as follows.



This is the truth table for CLR:

CH RDY	PW STROBE	X	RESET	Y	CLR
0	0	0	0	1	0
0	1	1	0	1	0
1	0	1	0	1	1
1	1	1	0	1	1
			1	0	0

For flip-flop FFA_W to turn "on", CLR must be high; for CLR to be high, RESET must be low, and either CH RDY or PW STROBE must be high.

Now CH RDY provides FFA_W with its data (D) input, and PW STROBE provides the clock (C) input. Therefore the function table for flip-flop FFA_W may be illustrated as follows:

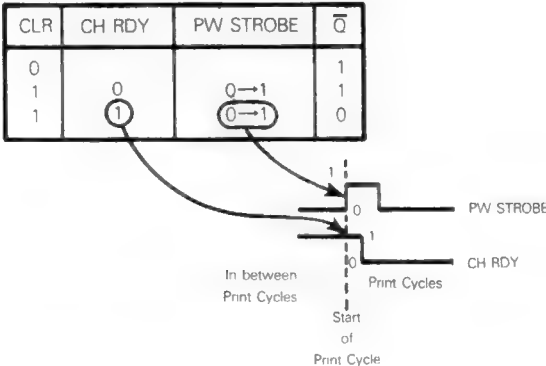
Inputs				Outputs		
PRESET	CLR	CLOCK (PW STROBE)	D (CH RDY)	Q	\bar{Q}	
0	1	0 or 1	0 or 1	1	0	PRESET = 1
1	0	0 or 1	0 or 1	0	1	
0	0	0 or 1	0 or 1	Unstable		PRESET = 1
1	1	0 \rightarrow 1	1	1	0	
1	1	0 \rightarrow 1	0	0	1	
1	1	0	0 or 1	Previous Q	Previous \bar{Q}	No change

And this devolves to the following small function table:

CLR	CH RDY	PW STROBE	\bar{Q}	
0			1	} possible "on" conditions
1	0	0 → 1	1	
1	1	0 → 1	0	

It takes a 0 to 1 transition of PW STROBE for flip-flop FFA_W to turn on. When FFA_W turns on, however, if CH RDY is 0, then the \bar{Q} output is still 1, representing the "off" condition. Thus, to turn FFA_W "on", PW STROBE must go from 0 to 1 while CH RDY is 1.

Recall that CH RDY is a signal which is output high in between print cycles and is output low for the duration of a print cycle. This means that flip-flop FFA_W will only turn on if PW STROBE pulses high in between print cycles, as characterized by CH RDY being output high:



For the moment do not worry about how CH RDY goes to 0 shortly after flip flop FFA_W turns on. We will explain how this happens later. The only important thing to note is that a PW STROBE high pulse will be ignored if it occurs while CH RDY is low.

What about the RESET signal? What this signal does is override all other logic associated with flip-flop FFA_W; whenever RESET is input high, CLR is forced low which turns flip-flop FFA_W off irrespective of whatever else is going on.

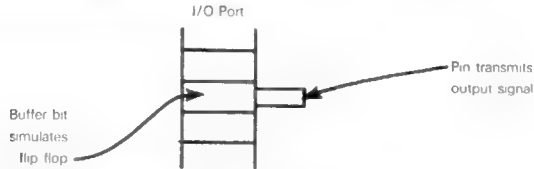
RESET

SIMULATING FLIP-FLOP FFA_W

We concluded in Chapter 2 that a flip-flop is represented in a microcomputer system by a single bit of read/write memory. A single bit of a read/write buffer will do just as well.

I/O Port 1 has been assigned to output signals. This port has an 8-bit buffer to which port pins are connected; thus **each bit of the port buffer will simulate the flip-flop whose output is transmitted via the port pin:**

**FLIP-FLOP
SIMULATION
USING I/O
PORTS**



Recall that FFA has been assigned pin 0 of I/O Port 1.

O.K., we are ready to simulate flip-flop FFA_W.

At the same time, how about simulating the three gates below and to the left of FFA_W? These three gates are numbered 26, 27, and 37 and together they create the CLR input.

Simulating these three gates individually, the following instruction sequence applies:

```

;SIMULATE GATE 27
IN      2          ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
CMP     A,B        ;COMPLEMENT ALL EIGHT BITS
ANI     40H        ;ISOLATE BIT 6; IT REPRESENTS RESET COMPLEMENT
MOV     B,A        ;SAVE COMPLEMENT IN REGISTER B

;SIMULATE GATE 26
IN      2          ;INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR
ANI     22H        ;ISOLATE BITS 5 AND 1; THEY REPRESENT
MOV     C,A        ;PW STROBE AND CH RDY. SAVE IN CPU REGISTER C

;SIMULATE GATE 37
JZ      CLR0       ;IF BITS 1 OR 5 DO NOT EQUAL 1, CLR IS 0
MOV     A,B        ;TEST COMPLEMENT OF RESET BY MOVING
AND     A,A        ;TO ACCUMULATOR AND ANDING WITH ITSELF
JZ      CLR0       ;IF RESULT IS 0, CLR IS 0
MVI     D,1        ;CLR IS 1 SO STORE 1 IN D, BIT 0
JMP     FFAW       ;

CLR0    MVI     D,0 ;CLR IS 0 SO STORE 0 IN D, BIT 0

;SIMULATE FLIP-FLOP FFAW
FFAW    MOV     A,D ;TEST STATUS OF CLR BY MOVING D TO
RRC     ;ACCUMULATOR AND SHIFTING BIT 0 INTO CARRY
JNC     FFA0       ;IF CLR IS 0, SET I/O PORT 1, BIT 0 TO 1
MOV     A,C        ;CLR IS NOT 0. TEST PW STROBE
ANI     20H        ;IF PW STROBE IS 0, CLOCK HAS NOT PULSED
  
```

```

JZ      FFA0      ;SET BIT 0 OF I/O PORT 1 TO 1
MOV     A,C       ;PW STROBE IS 1, TEST CH RDY
ANI     02H       ;IF CH RDY IS 0, SET BIT 0 OF I/O PORT 1 TO 0
JZ      FFA0
IN      1         ;LOAD I/O PORT 1 INTO ACCUMULATOR
ANI     F7H       ;BIT 0 MUST BE RESET TO 0, SINCE FFA IS "ON"
OUT     1
JMP     FFB       ;JUMP TO FLIP-FLOP B SIMULATION
FFA0    IN      1  ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ORI     1  ;BIT 0 MUST BE SET TO 1 SINCE FFA IS "OFF"
        OUT     1

```

;FLIP-FLOP FFB SIMULATION FOLLOWS

It is very important that you understand how instructions fit together to make a program. Read no further until you understand completely how the instruction sequence given above simulates the logic of FFA_W and its three associated gates.

Let us look at the above simulations.

The RESET signal, you will recall, has been tied to bit 6 of the 8255 I/O Port C, this port is addressed as I. O Port 2 based on the way in which we have elected to wire the 8255 PPI into our microcomputer system. In order to invert this signal, we input the contents of I/O Port 2 to the Accumulator, complement the contents of the Accumulator, then isolate the complement of RESET by setting all bits of the Accumulator to 0, bar bit 6

from I/O Port 2

```

IN      2      XXXXXXXX to Accumulator
CMP     XXXXXXXX Complement
ANI     40H    01000000 Isolate bit 6
          0X000000

```

The complement of RESET is saved temporarily in CPU Register B. **The simulation of gate 27 is complete.**

The simulation of gate 26 is not quite as straightforward.

We are seeking the OR of PW STROBE and CH RDY. These two signals are represented by bits 5 and 1, respectively, of I/O Port 2. Now what we do is load the contents of I/O Port 2 into the Accumulator, then execute an ANI instruction which sets all bits to 0, bar bits 5 and 1. But we do not actually OR these two remaining bits. Why? The reason is because when the ANI instruction is executed, it sets the Zero status to the complement of (PW STROBE) OR (CH RDY):

**INVERTER
SIMULATION**

**OR GATE
SIMULATION**

**STATUS FLAGS
USED TO
REPRESENT
LOGIC**

A5 OR A1	Accumulator Contents								Hex Value	Zero Status
	A7	A6	A5	A4	A3	A2	A1	A0		
0	0	0	0	0	0	0	0	0	00	1
1	0	0	0	0	0	0	1	0	02	0
1	0	0	1	0	0	0	0	0	20	0
1	0	0	1	0	0	0	1	0	22	0

PW STROBE

CH RDY

Following ANI instruction execution, Zero status is complement of PW STROBE OR CH RDY

We can therefore move on to gate 37.

The purpose of gate 37 is to generate the $\overline{\text{FFA}}_{\text{FF}}$ CLR input. We are going to simulate CLR using bit 0 of CPU Register D. Now we come right out of the gate 26 simulation into the gate 37 simulation; at this time the Zero status will be 0 if the OR of PW STROBE and CH RDY is 1; Zero status will be 1 otherwise. (Recall that Zero statuses always represent the inverse of the 0 condition. In other words, a 0 condition causes the Zero status to be set to 1, a nonzero condition causes the Zero status to be set to 0.)

**ZERO
STATUS**

The first instruction of the gate 37 simulation takes advantage of the fact that we have the OR of PW STROBE and CH RDY recorded in the Zero status. If the Zero status is 1, CLR must be 0, so the first JZ instruction branches to logic that will set bit 0 of CPU Register D to 0. The next group of instructions in the gate 37 simulation test the complement of RESET, as stored in CPU Register B, by moving Register B contents to the Accumulator then ANDing the Accumulator with itself. This AND will not change the contents of the Accumulator, but it will reset statuses based on the result of the AND. If the complement of RESET is 0, then the JZ instruction which follows will branch to program logic which sets bit 0 of CPU Register D to 0. If the complement of RESET is not 0, then all conditions have been met for gate 37 to output a nonzero result — and this condition is simulated by the MVI D,1 instruction, which sets bit 0 of CPU Register D to 1.

Flip-flop FFA is simulated next. The state of this flip-flop may be defined as follows

If CLR is 0 then \overline{Q} is 1.

If PW STROBE is 0 then \overline{Q} is 1.

If CLR is 1 and PW STROBE is 1 and CH RDY is 0 then \overline{Q} is 1.

If CLR is 1 and PW STROBE is 1 and CH RDY is 1 then \overline{Q} is 0.

CLR is simulated by bit 0 of the D register. PW STROBE is simulated by bit 5 of the C register. CH RDY is simulated by bit 1 of the C register.

The simulation of flip-flop FFA begins with the instruction labeled $\overline{\text{FFA}}_{\text{FF}}$.

First we test the status of CLR by moving the D Register contents to the Accumulator and shifting bit 0 into the Carry status. These two steps are necessary since 8080 shift instructions only work on the contents of the Accumulator. If the Carry is 0, then bit 0 of the D Register must be 0, which means that CLR is 0, therefore, we branch to instructions which set bit 0 of I/O Port 1 to 1. (Observe that \overline{Q} is being simulated, which is why, for the "off" condition, we set bit 0 of I/O Port 1 to 1, instead of resetting it to 0.)

**CARRY
STATUS**

Presuming that CLR has a value of 1, we next test PW STROBE. To do this we move the contents of the C Register to the Accumulator, then isolate the PW STROBE bit by ANDing with an appropriate mask. If PW STROBE is 0, we must again branch to instructions which set bit 0 of I/O Port 1 to 1.

Assuming that PW STROBE is 1, all that remains is to check the condition of CH RDY. To do this, we again move the contents of the C Register to the Accumulator, then isolate CH RDY by ANDing with an appropriate mask. If CH RDY is 0, we branch to instructions which set bit 0 of I/O Port 1 to 1.

Assuming that all conditions have been met to turn flip-flop FFA on, we must set bit 0 of I/O Port 1 to 0. This is done by inputting the contents of I/O Port 1 to the Accumulator, ANDing with the appropriate mask, then returning the result:

7 6 5 4 3 2 1 0	Bit No.
X X X X X X X Y	Accumulator contents
1 1 1 1 1 1 1 0	F7H
X X X X X X X 0	AND

The last three instructions of the flip-flop FFA simulation are the three instructions which set bit 0 of I/O Port 1 to 1 (reflecting the fact that flip-flop FFA is "off".) These three instructions load the contents of I/O Port 1 into the Accumulator, OR with the appropriate mask, then return the result.

SWITCHING A BIT ON

7 6 5 4 3 2 1 0	Bit No.
X X X X X X X Y	Accumulator contents
<u>0 0 0 0 0 0 0 1</u>	1
X X X X X X X 1	OR

Now in all honesty, the program sequence we have just described is a ridiculous way of simulating flip-flop FFA and its three associated gates.

It is ridiculous because we simulated each gate as an independent transfer function. Instead, let us consider the flip-flop, with its three gates, as a single transfer function. We can represent the transfer function with the following state definition:

Set \bar{Q} to 0 if RESET = 0, CH RDY = 1 and PW STROBE goes from 0 to 1. Set \bar{Q} to 1 otherwise.

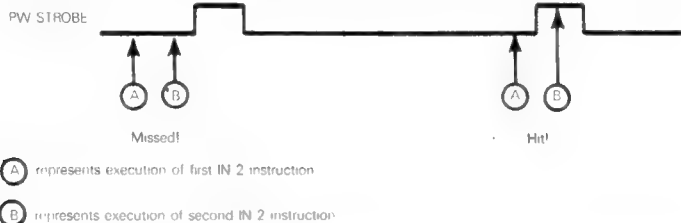
How are we going to test for the transition of PW STROBE from 0 to 1?

Using interrupts, the test would be very simple; but we are not going to use interrupts until Chapter 5

Without using interrupts, there is only one way to check for a PW STROBE 0 to 1 transition. We must input the contents of I/O Port 2 to the Accumulator, isolate bit 5, save the result, input the contents of I/O Port 2 to the Accumulator again, isolate bit 5

SIGNAL LEVEL CHANGES SENSED WITHOUT INTERRUPTS

again, then compare the two bits for an old value of 0 and a new value of 1. But this scheme is risky; it will only catch signal transitions which are lucky enough to occur in between the two instructions which load I/O Port 2 contents to the Accumulator.



Within the logic of a microcomputer program, however, we have no need to rely on signal transitions. Event sequences are determined by instruction execution sequence. The whole concept of timing on the leading or trailing edge of a signal pulse has no meaning. Instead of using PW STROBE signal transitions, therefore, we will use PW STROBE signal levels. Flip-flop FFA can now be described with the following state definition:

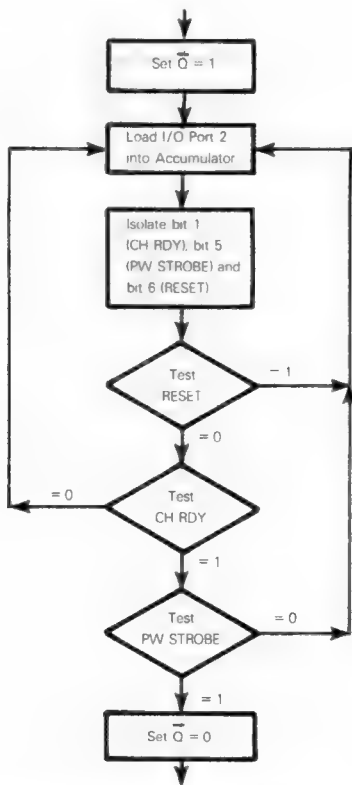
EVENT TIMING IN MICROCOMPUTER SYSTEM

Set \bar{Q} to 0 if RESET equals 0, CH RDY equals 1 and PW STROBE equals 1. Set \bar{Q} to 1 otherwise.

If you are a logic designer, you may be deeply troubled by the blithe way in which we simply replace edge triggering with level triggering. We can do this within a microcomputer system because microcomputer programming gives us an extra degree of freedom, as compared with digital logic design: The order in which you stuff logic components into a PC card has nothing to do with the sequence in which logical events occur. Logic sequence is going to be controlled by edge and level triggering. But the order in which you write assembly language instructions is the order in which the instructions will be executed.

TIMING AND LOGIC SEQUENCE

To drive this point home, look at the following flow chart which represents the state definition for flip-flop FFA:



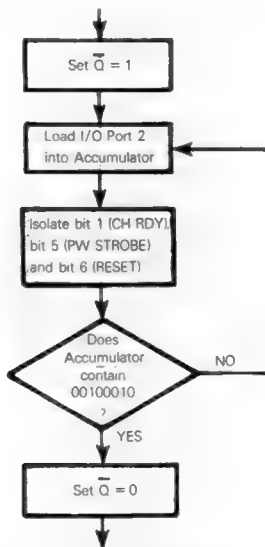
Each rectangular box represents a data movement or manipulation operation.

Each diamond represents logic which tests the condition of a status flag.

The order in which you write down instructions is the order in which instructions will be executed. With regard to the flow chart above, this execution sequence is represented by the continuous line of downward pointing arrows. Special Jump-On-Condition instructions allow the normal sequence to be modified, as represented by the horizontal arrows emanating from the sides of the diamonds. You can follow the arrows to the point where the Jump-On-Condition instruction takes you.

We will now rewrite the flip-flop FFA simulation treating the flip-flop and the three CLR logic gates as a single transfer function.

Since RESET, CH RDY and PW STROBE are all connected to pins of I/O Port 2, we load the contents of I/O Port 2 into the Accumulator and isolate all three bits. Now there is only one combination of values that these three bits can have if a new print cycle is to begin. RESET must equal 0, while CH RDY and PW STROBE both equal 1. We will therefore redraw our program flow chart as follows:



Our instruction sequence condenses to the following few instructions:

:SIMULATION OF FFA AND ASSOCIATED LOGIC

IN	1	:INITIALLY SET BIT 0 OF I/O PORT 1 TO 1
ORI	1	
OUT	1	

:LOAD I/O PORT 2 CONTENTS INTO ACCUMULATOR

:AND ISOLATE BITS 1, 5 AND 6 FOR CH RDY,

:PW STROBE AND RESET, RESPECTIVELY

L10	IN	2	:INPUT I/O PORT 2 TO ACCUMULATOR
	ANI	62H	:ISOLATE BITS 6, 5 AND 1
	CPI	22H	:IF RESET = 0, CH RDY = 1 AND
	JNZ	L10	:PW STROBE = 1, NEW PRINT CYCLE STARTS
	IN	1	:OTHERWISE RETURN TO L10. START NEW
	ANI	FEH	:PRINT CYCLE BY SETTING I/O PORT 1, BIT 0 TO 0
	OUT	1	

:NEW PRINT CYCLE INSTRUCTION SEQUENCE STARTS HERE

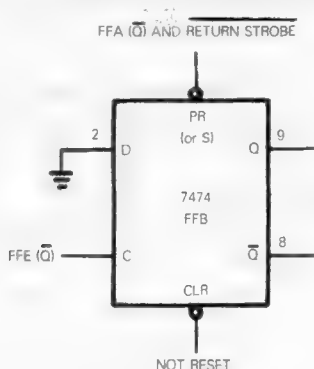
The first three instructions in the above sequences simply set bit 0 of I/O Port 1 to 1. This is in anticipation of a new print cycle not beginning. Four instructions, beginning with the instruction labeled L10, are all that is needed to check for conditions which trigger the start of a new print cycle. These four instructions execute in 34 clock cycles which, assuming a 500 nanosecond clock, means that PW STROBE must pulse high for at least 17 microseconds.

Providing RESET equals 0 while CH RDY and PW STROBE equals 1, a new print cycle must begin, so the last three instructions set bit 0 of I/O Port 1 to 0.

Our simulation of flip-flop FFA is complete.

FLIP-FLOP FFB_W

The next device in our logic sequence is another 7474 flip-flop, marked FFB_W in Figure 3-1; it is just to the right of FFA_W. This flip-flop may be illustrated as follows:



The following function table describes FFB, as wired above, with its D input tied to 0.

FFA (\bar{Q})	RETURN STROBE	PRESET	NOT RESET (CLR)	FFE (\bar{Q}) =CLOCK	Q	\bar{Q}
0	0	0	1	X	1	0
0	1	0	0	X	unstable	
1	0	0				
1	1	1	0	X	0	1
		1	1	0 \rightarrow 1	0	1

Chapter 2 provides the standard 7474 flip-flop function table; all we have done is remove the D column, and the rows that show D = 1. We can also remove the CLR column, and all rows that show CLR = 0, since CLR is tied to NOT RESET. NOT RESET will always be 1 within a print cycle, since FFA will not turn on if NOT RESET is 0.

The following simplified function table can now be used for FFB, assuming that CLR (NOT RESET) will always be 1 and D will always be 0:

FFA (\bar{Q}) AND RETURN STROBE =PRESET	FFE (\bar{Q}) =CLOCK	Q	\bar{Q}
0	0 or 1	1	0
1	0 \rightarrow 1	0	1

Let us take a look at the FFB PRESET input; it is FFA (\bar{Q}) AND RETURN STROBE.

RETURN STROBE, recall is a signal input by external logic to initiate a special print cycle which moves the printwheel back to its position of visibility, but does not fire the printhead, or print a character. We call this a "Printwheel Repositioning" print cycle. In between print cycles, therefore, RETURN STROBE must be input high.

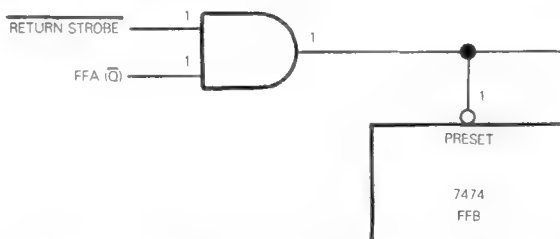
PRINTWHEEL REPOSITIONING PRINT CYCLE

Since RETURN STROBE is input low as an alternative method of initiating a print cycle, when simulating FFB, we are going to have to consider RETURN STROBE two ways:

- 1) As a contributor to the PRESET input.
- 2) As a signal which can initiate a print cycle, bypassing flip-flop FFA.

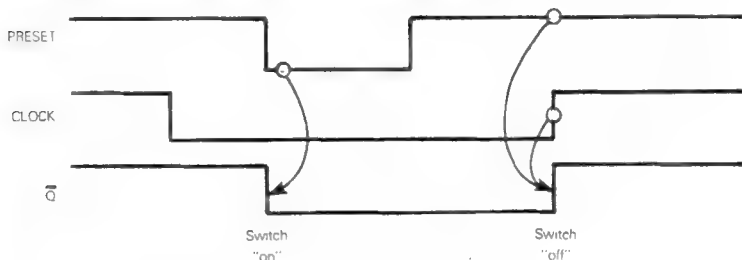
But first, let us define the condition of flip-flop FFB in between print cycles.

As we have just seen in our simulation of flip-flop FFA, the FFA (\bar{Q}) output is high until the beginning of a print cycle, when \bar{Q} goes low, the FFA (\bar{Q}) output is therefore high in between print cycles. By definition, RETURN STROBE is high in between print cycles, since RETURN STROBE low is used to initiate a printwheel repositioning print cycle. Therefore, the FFB PRESET input will be high in between print cycles:



Since PRESET is input high in between print cycles, we are going to assume that at the beginning of a print cycle FFB is off: that is, Q is output low and \bar{Q} is output high. This also assumes that at some recent time PRESET was input high when the \bar{Q} output of flip-flop FFE went from 0 to 1. As you will see later on, this is indeed what happens at the end of every print cycle.

Coming into a new print cycle, therefore, FFB has a high PRESET input, with a high \bar{Q} output and a low Q output. This flip-flop now acts as a switch: it is turned on by PRESET being input low; it is subsequently turned off by a clock 0 to 1 transition occurring after PRESET has again gone high:



The switch "on" illustrated above occurs under two circumstances:

- 1) Immediately after the onset of a new print cycle, when FFA outputs \bar{Q} low, thus forcing PRESET low.
- 2) When RETURN STROBE is input low signaling a printwheel repositioning print cycle

The switch "off" occurs when the FFE (\bar{Q}) output makes a low to high transition while PRESET is being input high; this occurs at the end of every print cycle.

SIMULATING FLIP-FLOP FFB

Bit 1 of I/O Port 1 has been assigned to the Q output of flip-flop FFB. The switch "on" illustrated above is therefore simulated by the following three instructions:

**SWITCHING
BITS ON**

```
IN      1      ;LOAD FLIP-FLOP DATA BYTE
ANI     FDH    ;RESET BIT 1 TO 0
OUT     1      ;RESTORE FLIP-FLOP DATA BYTE
```

This is how the ANI instruction works:

7 6 5 4 3 2 1 0	Bit No.
XXXXXXXYX	Accumulator contents
<u>1 1 1 1 1 0 1</u>	FDH
XXXXXX0X	AND

**SWITCHING
BITS OFF**

Subsequently the switch "off" will be simulated as follows:

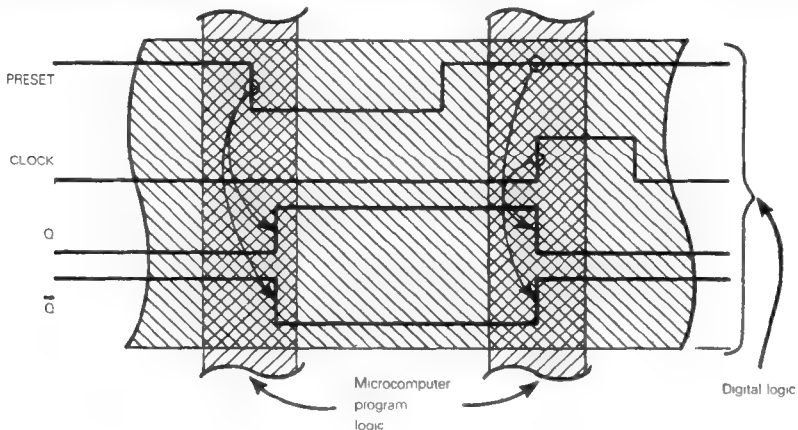
```
IN      1      LOAD FLIP-FLOP DATA BYTE
ORI     2      ;SET BIT 1 TO 1
OUT     1      ;RESTORE FLIP-FLOP DATA BYTE
```

This is how the ORI instruction works:

7 6 5 4 3 2 1 0	Bit No.
XXXXXXXYX	Accumulator contents
<u>0 0 0 0 0 1 0</u>	2
XXXXXX1X	OR

We now encounter a situation where, with every best intention, we are not going to be able to directly simulate our digital logic.

It is easy enough to draw one 7474 flip-flop in a logic diagram and connect its pins to suitable signals. Having done that, you no longer need to worry about when a signal does, or does not change state. Unfortunately, an assembly language instruction sequence has no pins or signals; **assembly language will simulate events that are occurring at one instant at a time only. For flip-flop FFB, this may be illustrated as follows:**



Immediately after flip-flop FFA turns on to usher in a new print cycle, it outputs \bar{Q} low, which in turn switches flip-flop FFB on. FFB will not switch off until some point much later in the print cycle when FFE outputs \bar{Q} high. **We must therefore divide our simulation of FFB into two parts:**

- 1) At the beginning of our program we will simulate FFB switching on, since chronologically it is the next event within the print cycle.
- 2) Later on in the program, when we simulate FFE setting \bar{Q} high, we must remember to simulate FFB switching off.

But that is not all there is to the FFB simulation. **We must also modify the instruction sequence that executes in between print cycles, so that RETURN STROBE input low can be simulated initiating a printwheel repositioning print cycle.**

With modified or new instructions shaded, this is how our program now looks:

```

;IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
;INITIALLY SET I/O PORT 1 BIT 1 TO 0, BIT 0 TO 1
      IN      1      ;INPUT I/O PORT 1 TO ACCUMULATOR
      ORI     1      ;SET BIT 0
      ANI     FDH     ;RESET BIT 1
      OUT     1      ;RETURN RESULT

;TEST FOR RETURN STROBE LOW
L10    IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
      ANI     10H     ;ISOLATE RETURN STROBE
      JZ      FFB     ;IF IT IS 0, JUMP TO FFB SIMULATION

;SIMULATION OF FFA AND ASSOCIATED LOGIC
;LOAD I/O PORT 2 CONTENTS INTO ACCUMULATOR AND
;ISOLATE BITS 1, 5 AND 6 FOR CH RDY, PW STROBE
;AND RESET, RESPECTIVELY
      IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
      ANI     62H     ;ISOLATE BITS 6, 5 AND 1
      CPI     22H     ;IF RESET = 0, CH RDY = 1 AND PW STROBE = 1
      JNZ     L10     ;START NEW PRINT CYCLE, OTHERWISE RETURN TO L10
      IN      1      ;TO START NEW PRINT CYCLE, SET
      ANI     FEH     ;I/O PORT 1, BIT 0 TO 0
      OUT     1

;NEW PRINT CYCLE SEQUENCE STARTS HERE
;SIMULATE FLIP-FLOP SWITCHING ON
FFB    IN      1      ;LOAD I/O PORT 1 INTO ACCUMULATOR
      ANI     FDH     ;RESET BIT 1 TO 0
      OUT     1      ;RESTORE RESULT

```

We are not quite finished with our simulation of flip-flop FFB. Observe that the \bar{Q} output from FFB goes to:

- 1) A 7411 AND gate, located approximately at co-ordinate B6.
- 2) A 7432 OR gate, located at A7.

The FFB (Q) output is not idle either, but we will look into it later.

First consider the 7411 AND gate located at B6.

If you refer back to the description of output signals, you will notice that CH RDY was declared to be high in between print cycles, but low during a print cycle.

In reality, CH RDY is output by the 7411 AND gate located at B6, therefore, in between print cycles, all three inputs to this AND gate must be high. Our analysis of flip-flop FFB shows that its Q output will indeed be high in between print cycles, but for the moment you must take it on faith that the other two signals input to the AND gate will also be high in between print cycles.

In any event, as soon as flip-flop FFB switches on, its \bar{Q} output goes low, which means that no matter what the other two inputs to the 7411 AND gate do, CH RDY will also be driven low. This change in the status of CH RDY is simulated by adding the following two instructions to our program:

:TEST FOR RETURN STROBE LOW

```
L10      IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
          ANI      10H    :ISOLATE RETURN STROBE
          JZ       FFB    :IF IT IS 0, JUMP TO FFB SIMULATION
```

:SIMULATION OF FFA AND ASSOCIATED LOGIC

:LOAD I/O PORT 2 CONTENTS INTO ACCUMULATOR AND

:ISOLATE BITS 1, 5 AND 6 FOR CH RDY, PW STROBE

:AND RESET, RESPECTIVELY

```
          IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
          ANI      62H    :ISOLATE BITS 6, 5 AND 1
          CPI      22H    :IF RESET = 0, CH RDY = 1 AND PW STROBE = 1,
          JNZ      L10    :START NEW PRINT CYCLE. OTHERWISE RETURN TO L10
          IN      1      :TO START NEW PRINT CYCLE, SET
          ANI      FEH    :I/O PORT 1, BIT 0 TO 0
          OUT      1
```

:NEW PRINT CYCLE SEQUENCE STARTS HERE

:SIMULATE FLIP-FLOP FFB SWITCHING ON

```
FFB      IN      1      :LOAD I/O PORT 1 INTO ACCUMULATOR
          ANI      FDH    :RESET BIT 1 TO 0
          OUT      1      :RESTORE RESULT
```

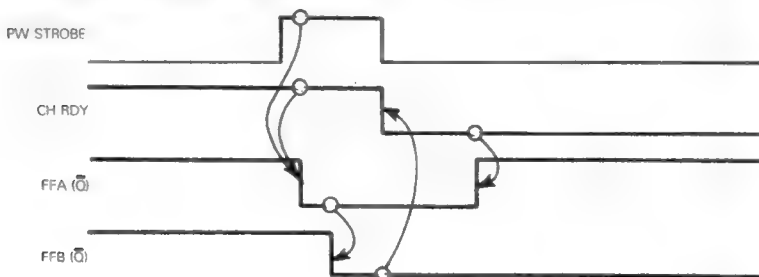
:SIMULATE 7411 AND GATE SWITCHING CH RDY LOW

```
          IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
          ANI      FDH    :RESET BIT 1 TO 0
          OUT      2      :RESTORE RESULT
```

We are now faced with an interesting problem. CH RDY becomes the D input to flip-flop FFA and it contributes to the CLR input of FFA. **What happens when CH RDY goes low in response to FFB switching on?**

Notice that PW STROBE only pulses high; therefore the OR gate located at co-ordinate B2 relies on CH RDY being high in order to provide a high input to the following AND gate. This AND gate, in turn, provides a high CLR input to flip-flop FFA. In other words, by the time flip-flop FFB turns "on" and switches CH RDY low, PW STROBE will have already gone low; thus inputs PW STROBE and CH RDY will both be low. **If you look back at flip-flop FFA's CLR truth table, you will find that when CH RDY and PW STROBE are both 0, CLR will always be 0.**

Therefore flip-flop FFA will switch off:



What does this mean? Our conclusion is that flip-flop FFA switches itself "on" at the beginning of a print cycle, but only stays on long enough to switch flip-flop FFB "on". When FFB turns "on", it sets CH RDY low, and that turns flip-flop FFA "off".

But here is the rub: if you look again at Figure 3-1, you will find that flip-flop FFA helps generate the J input to flip-flop FFC, in addition to switching on flip-flop FFB.

TIMING AND LOGIC SEQUENCE

Now that events are serialized in time, we can go ahead and simulate flip-flop FFA being turned "off", so long as we remember, when simulating flip-flop FFC, that it receives \bar{Q} low from flip-flop FFA. Bearing this precaution in mind, we will extend our program as follows:

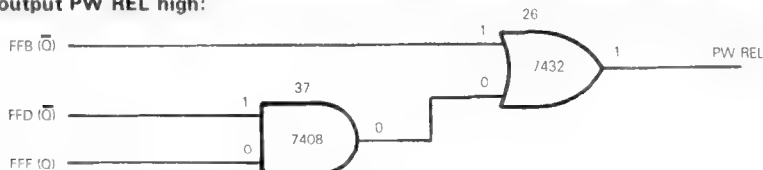
```

:TEST FOR RETURN STROBE LOW
L10      IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
          ANI     10H    :ISOLATE RETURN STROBE
          JZ      FFB    :IF IT IS 0, JUMP TO FFB SIMULATION

;SIMULATION OF FFA AND ASSOCIATED LOGIC
:LOAD I/O PORT 2 CONTENTS INTO ACCUMULATOR AND
:ISOLATE BITS 1, 5 AND 6 FOR CH RDY, PW STROBE
:AND RESET, RESPECTIVELY
          IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
          ANI     62H    :ISOLATE BITS 6, 5 AND 1
          CPI     22H    :IF RESET = 0, CH RDY = 1 AND PW STROBE = 1,
          JNZ     L10    :START NEW PRINT CYCLE. OTHERWISE RETURN TO L10
          IN      1      :TO START NEW PRINT CYCLE, SET
          ANI     FEH    :I/O PORT 1, BIT 0 TO 0
          OUT     1
          :NEW PRINT CYCLE SEQUENCE STARTS HERE
          :SIMULATE FLIP-FLOP FFB SWITCHING ON
          FFB      IN    1      :LOAD I/O PORT 1 INTO ACCUMULATOR
          ANI     FDH    :RESET BIT 1 TO 0
          OUT     1      :RESTORE RESULT
          :SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
          IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
          ANI     FDH    :RESET BIT 1 TO 0
          OUT     2      :RESTORE RESULT
          :CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1
          IN      1      :LOAD I/O PORT 1 INTO ACCUMULATOR
          ORI     1      :SET BIT 0 TO 1
          OUT     1      :RESTORE RESULT

```

Now look at the OR gate located at co-ordinate A7. This gate receives the FFB \bar{Q} output as one of its inputs in order to generate PW REL. The other input to this OR gate is the AND of the Q output from flip-flop FFF, plus the \bar{Q} output of flip-flop FFD. You will find out shortly that these flip-flops are also turned "off" in between print cycles; they are turned on sequentially during the course of the print cycle. At the point where FFB switches on, FFF will be switched off, which means that its Q output will be low; thus, the AND gate located at A6 will output low, which means that OR gate 26 has been relying on the high \bar{Q} output from FFB in order to output PW REL high:



Now, when FFB switches "on" and outputs \bar{Q} low, PW REL will also output low. We must therefore modify our program to output bits 0 and 1 of I/O Port 3 low, since both PW REL and CH RDY are both going to be driven low. This is how our program now looks:

```

;TEST FOR RETURN STROBE LOW
L10  IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
      ANI     10H    ;ISOLATE RETURN STROBE
      JZ      FFB     ;IF IT IS 0, JUMP TO FFB SIMULATION

;SIMULATION OF FFA AND ASSOCIATED LOGIC
;LOAD I/O PORT 2 CONTENTS INTO ACCUMULATOR AND
;ISOLATE BITS 1, 5 AND 6 FOR CH RDY, PW STROBE
;AND RESET, RESPECTIVELY
      IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
      ANI     62H    ;ISOLATE BITS 6, 5 AND 1
      CPI     22H    ;IF RESET = 0, CH RDY = 1 AND PW STROBE = 1
      JNZ     L10    ;START NEW PRINT CYCLE. OTHERWISE RETURN TO L10
      IN      1      ;TO START NEW PRINT CYCLE. SET
      ANI     FEH    ;I/O PORT 1, BIT 0 TO 0
      OUT     1
;NEW PRINT CYCLE SEQUENCE STARTS HERE
;SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB  IN      1      ;LOAD I/O PORT 1 INTO ACCUMULATOR
      ANI     FDH    ;RESET BIT 1 TO 0
      OUT     1      ;RESTORE RESULT
;SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
;ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW
      IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
      ANI     FCH    ;RESET BITS 0 AND 1 TO 0
      OUT     2      ;RESTORE RESULT
;CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1
      IN      1      ;LOAD I/O PORT 1 INTO ACCUMULATOR
      ORI     1      ;SET BIT 0 TO 1
      OUT     1      ;RESTORE RESULT

```

Do we have to do anything about the Q output from flip-flop FFB? If you look at this output you will see that it ties directly to the RESET inputs of flip-flops FFC, FFD, and FFE. It also becomes one of the inputs to the 555 multivibrator.

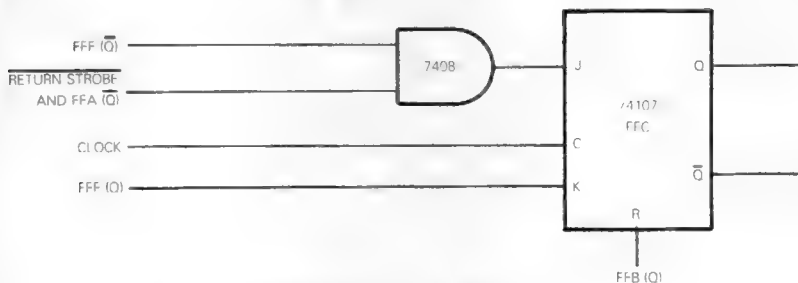
In fact, the FFB Q output is a clamping signal, when low, it shuts the four connected devices off; when high, these four devices are switched on.

The FFB Q output will be taken into account when we simulate the four devices connected to this signal. Therefore, our simulation of flip-flop FFB is done.

FLIP-FLOP FFC

This is the 74107 flip-flop at co-ordinate C2 in Figure 3-1. Since we are going to simulate four 74107 flip-flops, you should refer back to Chapter 2 if you cannot immediately recall the characteristics of this device.

Let us isolate flip-flop FFC to see how it works:



INPUTS				OUTPUTS	
R	C	J	K	Q	\bar{Q}
L	X	X	X	L	H
H		L	L	stay 1	stay 0
H		H	L	0	1
H		L	H	1	0
H		H	H	invert	invert

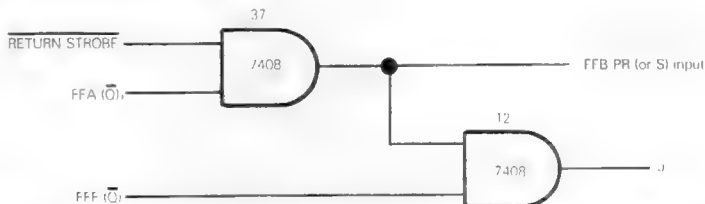


In between print cycles, the Q output of FFB, being low, switches flip-flop FFC off. FFC, therefore, outputs Q low and \bar{Q} high.

What happens when FFB is switched on depends on the J and K inputs arriving at FFC.

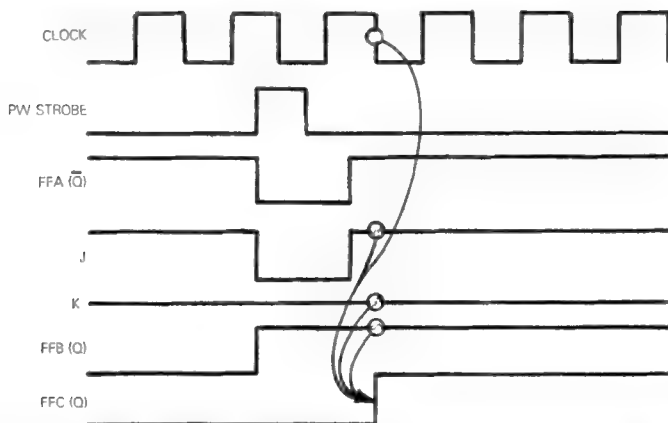
In between print cycles flip-flop FFF is switched off, therefore its Q output will be low. FFC receives its K input from the FFF Q output, therefore when FFC switches on its K input will be 0.

The J input to FFC is generated as follows:



FFF (\bar{Q}) will be high, since FFF is switched off. The FFC J input will therefore be identical to the FFB PR input, which we have already described.

In summary, this is the signal sequence which turns FFC on:



When the FFB Q output goes high, unclamping FFC, FFC waits until the FFA \bar{Q} output goes high again; then FFC will receive a high input at J and a low input at K. On the trailing edge of the next clock pulse input to FFC, Q will be output high and \bar{Q} will be output low.

FFC waits for the FFA \bar{Q} output to go high again, because while FFA is switched on, \bar{Q} is output low. While FFA (\bar{Q}) (or RETURN STROBE) is pulsed low, FFC receives a low J input. So long as FFC is receiving low J and K inputs, its outputs will not change — that is one of the properties of a 74107 flip-flop.

Flip-flop FFC will remain in its "on" state until some later point in the print cycle when flip-flop FFF switches on. At that time, flip-flop FFC will receive a high input at K and a low input at J; and that will cause FFC to switch off.

SIMULATING FLIP-FLOP FFC

The simulation of flip-flop FFC is indeed straightforward; it involves these three steps:

- 1) We must adjust our initialization instructions to ensure that flip-flop FFC is reported as "off" in between print cycles.
- 2) The flip-flop FFB simulation must be followed immediately by instructions which simulate flip-flop FFC turning on.
- 3) We must remember to simulate FFC turning off — but that will not happen until some later point in the program.

Now the following modifications to the beginning of our program insure that flip-flop FFC is simulated "off" in between print cycles:

IN BETWEEN PRINT CYCLES PROGRAM EXECUTION

INITIALLY SET I/O PORT BITS 2 AND 1 TO 0. BIT 0 TO 1

```

A → IN      1      ;INPUT I/O PORT 1 TO ACCUMULATOR
   ORI      1      ;SET BIT 0
   ANI      F0H    ;RESET BITS 2 AND 1
   OUT      1      ;RETURN RESULT

```

;TEST FOR RETURN STROBE LOW

```

L10 IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
    ANI      10H   ;ISOLATE RETURN STROBE
    JZ       FFB   ;IF IT IS 0, JUMP TO FFB SIMULATION

```

All we have done is modify the AND mask to include I/O Port bit 2 among the bits reset to 0:

		Accumulator	
		Contents	
		7 6 5 4 3 2 1 0	← Bit No.
IN	1	XXXXXX XX	
ORI	1	000000 1	
		XXXXXX XX	
ANI	F9H	111110 1	
		XXXXXX 01	

Recall that I/O Port bit 2 has been assigned to flip-flop FFC.

What about the time delay that separates flip-flops B and C switching on? Recall that flip flop FFC will not switch on until after flip-flop FFB has switched flip-flop FFA off. If this is a printwheel repositioning print cycle, then FFC will not switch on until RETURN STROBE is input high again.

**TIMING AND
LOGIC
SEQUENCE**

The simplicity or complexity of our timing problem depends entirely on logic beyond Figure 3-1. There is nothing within the logic of Figure 3-1 that demands a time delay of fixed duration or, for that matter, any time delay separating FFB and FFC switching on. We will therefore pay no attention to the timing considerations associated with FFC switching on, rather **we will simply add simulation instructions to the end of our program as follows:**

```

;NEW PRINT CYCLE SEQUENCE STARTS HERE
;SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB      IN      1      ;LOAD I/O PORT 1 INTO ACCUMULATOR
         ANI     FDH     ;RESET BIT 1 TO 0
         OUT     1      ;RESTORE RESULT

;SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
;ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW
         IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
         ANI     FCH     ;RESET BITS 0 AND 1 TO 0
         OUT     2      ;RESTORE RESULT

;CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1
         IN      1      ;LOAD I/O PORT 1 INTO ACCUMULATOR
         ORI     1      ;SET BIT 0 TO 1
         OUT     1      ;RESTORE RESULT

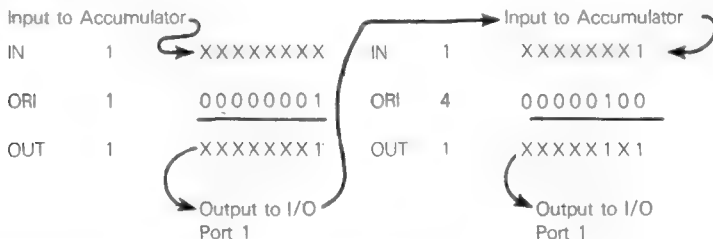
;SIMULATE 74107 FLIP-FLOP FFC SWITCHING ON
;SET BIT 2 OF I/O PORT 1 TO 1
         IN      1      ;LOAD I/O PORT 1 INTO ACCUMULATOR
         ORI     4      ;SET BIT 2 TO 1
         OUT     1      ;RESTORE RESULT

```



If you are beginning to think like a programmer, you will detect an opportunity for economy in the simulation of flip-flop FFC switching on. **Observe that the three instructions directly above (B) are also setting a bit of I/O Port 1 to 1.** This generates the following sequence of events:

**PROGRAMS
MADE
SHORTER**



We can combine the two operations as follows:

```

IN      1      XXXXXXXX
ORI     5      00000101
                XXXXX1X1
  
```

The instructions marked (B) now disappear, and are replaced by these modifications, marked (C):

```

;SIMULATE FLIP-FLOP FFB SWITCHING ON
FFB  IN      1      ;LOAD I/O PORT 1 INTO ACCUMULATOR
     ANI     FDH    ;RESET BIT 1 TO 0
     OUT     1      ;RESTORE RESULT

;SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
;ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW
     IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
     ANI     FCH    ;RESET BITS 0 AND 1 TO 0
     OUT     2      ;RESTORE RESULT

;CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1
;ALSO SIMULATE FFC TURNING ON. SET BIT 2 OF I/O PORT 1 TO 1
     IN      1      ;LOAD I/O PORT 1 INTO ACCUMULATOR
     ORI     5      ;SET BITS 2 AND 0 TO 1
     OUT     1      ;RESTORE RESULT
  
```

Our simulation of flip-flop C switching on is complete.

START RIBBON MOTION PULSE SIMULATION

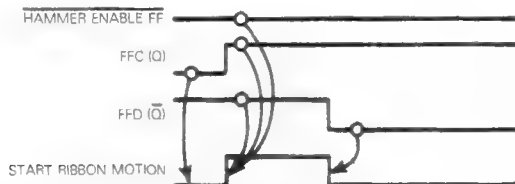
Recall that early in a print cycle the **START RIBBON MOTION** output signal is pulsed high to trigger external logic which advances the ribbon; thus when the printhead fires, fresh ribbon is in front of the character being printed. The **START RIBBON MOTION** signal is generated by a 7411 AND gate (number 7) located at co-ordinate C7 in Figure 3-1. This AND gate has three inputs:

- 1) **HAMMER ENABLE FF**. This is a signal input to identify a printwheel repositioning print cycle.
- 2) The Q output from flip-flop FFC.
- 3) The \bar{Q} output from flip-flop FFD.

HAMMER ENABLE FF will be high unless a printwheel repositioning print cycle is in progress, in which case the ribbon does not have to be moved. This signal, therefore, suppresses the **START RIBBON MOTION** pulse.

In between print cycles, flip-flops FFC and FFD are both switched off; therefore FFC (Q) is low and FFD (\bar{Q}) is high. **The FFC (Q) output holds the START RIBBON MOTION signal low.**

When FFC switches on during a normal print cycle, all inputs to AND gate 7 will be high, so START RIBBON MOTION will pulse high; it will stay high until flip-flop FFD switches on, at which time FFD will output \bar{Q} low; and that will drop START RIBBON MOTION pulse low. Timing may be illustrated as follows:



If you look at the timing diagram illustrated in Figure 3-2, you will see that the START RIBBON MOTION output pulse is extremely short. Therefore, instead of using flip-flop FFD to time the end of the START RIBBON MOTION HIGH PULSE, **we will simply execute instructions to turn bit 3 of I/O Port C on, then immediately turn it off, as follows:**

:NEW PRINT CYCLE SEQUENCE STARTS HERE

:SIMULATE FLIP-FLOP FFB SWITCHING ON

```
FFB    IN      1      :LOAD I/O PORT 1 INTO ACCUMULATOR
      ANI      FDH     :RESET BIT 1 TO 0
      OUT      1      :RESTORE RESULT
```

:SIMULATE 7411 AND GATE SWITCHING CH RDY LOW

:ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW

```
      IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
      ANI      FCH     :RESET BITS 0 AND 1 TO 0
      OUT      2      :RESTORE RESULT
```

:CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1

:ALSO SIMULATE FFC TURNING ON. SET BIT 2 OF I/O PORT 1 TO 1

```
      IN      1      :LOAD I/O PORT 1 INTO ACCUMULATOR
      ORI      5      :SET BITS 2 AND 0 TO 1
      OUT      1      :RESTORE RESULT
```

:PULSE START RIB MOTION HIGH

```
      IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
      ORI      8      :SET BIT 3 HIGH
      OUT      2      :OUTPUT RESULT
      ANI      F7H     :TURN BIT 3 OFF
      OUT      2      :OUTPUT RESULT
```

We can calculate the START RIB MOTION pulse width by adding the instruction execution times between pin 3 of I/O Port 2 being set high, then being reset low:

**PULSE WIDTH
CALCULATION**

Cycles	Instruction	
10	OUT 2	:OUTPUT RESULT
7	ANI F7H	:TURN BIT 3 OFF
10	OUT 2	:OUTPUT RESULT

Pulse width = 17 cycles, or 8.5
microseconds using a 500 nanosecond clock.

What happens next? Our logic sequence may take us to flip-flop FFD, to the right of FFC, or we may drop down to the 74121 one-shot number 36, just below and to the right of FFC.

One-shot 36 has its two A inputs tied to ground, which means that they will both input low. If you look at the 74121 function table given in Chapter 2, you will find that in this configuration, a one-shot output is triggered by a low-to-high transition at B. FFC (\bar{Q}) provides this trigger. Any other B input will keep this one-shot turned off — which means that **Q and \bar{Q} will output low and high, respectively, until much later in the print cycle, when FFC switches off;** that is when the FFC \bar{Q} output makes a low-to-high transition.

Flip-flop FFD becomes the next device to be simulated.

FLIP-FLOP FFD

Flip flop FFD receives its J input directly from the FFC (Q) output; it receives its K input from the FFC (\bar{Q}) output. Remember, since one-shot 36 is still switched off, its \bar{Q} output will be high; that means AND gate 12 will simply allow the FFC (\bar{Q}) output to propagate straight through, to become the FFD (K) input.

Now, flip-flop FFD receives the same reset and clock signals as FFC, therefore **flip-flop FFD will simply switch on one clock cycle later than flip-flop FFC.**

SIMULATING FLIP-FLOP FFD

The simulation of flip-flop FFD is almost identical to the simulation of flip-flop FFC; the principle difference is that bit 3 of I/O Port 1 has been assigned to flip-flop FFD. Once again, we are going to limit ourselves to switching flip-flop FFD on and ensuring that its setting in between print cycles is correct.

Flip-flop FFD is switched off later in the print cycle; we must therefore remember to switch off later in the program.

Here are the necessary program modifications and additions:

:IN BETWEEN PRINT CYCLES PROGRAM EXECUTION

:INITIALLY SET I/O PORT BITS 3, 2 AND 1 TO 0, BIT 0 TO 1

```

    IN      1      :INPUT I/O PORT 1 TO ACCUMULATOR
    ORI     1      :SET BIT 0
    ANI     F1H    :RESET BITS 3, 2, AND 1
    OUT     1      :RETURN RESULT

```

:TEST FOR RETURN STROBE LOW

```

L10    IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
       ANI     10H    :ISOLATE RETURN STROBE
       JZ      FFB     :IF IT IS 0, JUMP TO FFB SIMULATION

```

:SIMULATE FLIP-FLOP FFB SWITCHING ON

```

FFB    IN      1      :LOAD I/O PORT 1 INTO ACCUMULATOR
       ANI     FDH    :RESET BIT 1 TO 0
       OUT     1      :RESTORE RESULT

```

:SIMULATE 7411 AND GATE SWITCHING CH RDY LOW

:ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW

```

       IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
       ANI     FCH    :RESET BITS 0 AND 1 TO 0
       OUT     2      :RESTORE RESULT

```

:CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1

```

;ALSO SIMULATE FFC TURNING ON. SET BIT 2 OF I/O PORT 1 TO 1
IN      1      ;LOAD I/O PORT 1 INTO ACCUMULATOR
ORI     5      ;SET BITS 2 AND 0 TO 1
OUT     1      ;RESTORE RESULT

:PULSE START RIB MOTION HIGH
IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
ORI     8      ;SET BIT 3 HIGH
OUT     2      ;OUTPUT RESULT
ANI     F7H    ;TURN BIT 3 OFF
OUT     2      ;OUTPUT RESULT

;SIMULATE FFD TURNING ON. SET BIT 3 OF I/O PORT 1 TO 1
IN      1      ;LOAD I/O PORT 1 INTO ACCUMULATOR
ORI     8      ;SET BIT 3 TO 1
OUT     1      ;RESTORE RESULT

```

← (E)

If the program modifications and additions illustrated above are not immediately obvious, compare them to the flip-flop C simulation. Do not go on if you do not understand the flip flop FFD program changes

Just as the simulation of FFC switching on (B) was absorbed into the FFB simulation (C), so the simulation of FFD switching on (E) can be absorbed as follows:

**PROGRAMS
MADE
SHORTER**

```

;NEW PRINT CYCLE SEQUENCE STARTS HERE
;SIMULATE FLIP - FLOP FFB SWITCHING ON
FFB     IN      1      ;LOAD I/O PORT 1 INTO ACCUMULATOR
        ANI     FDH    ;RESET BIT 1 TO 0
        OUT     1      ;RESTORE RESULT

;SIMULATE 7411 AND GATE SWITCHING CH RDY LOW
;ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW
IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
ANI     FCH    ;RESET BITS 0 AND 1 TO 0
OUT     2      ;RESTORE RESULT

;CH RDY LOW TURNS FFA OFF. SET BIT 0 OF I/O PORT 1 TO 1
;ALSO SIMULATE FFC AND FFD TURNING ON. SET BITS 3 AND 2 OF I/O PORT 1 TO 1
IN      1      ;LOAD I/O PORT 1 INTO ACCUMULATOR
ORI     0DH    ;SET BITS 3, 2 AND 0 TO 1
OUT     1      ;RESTORE RESULT

:PULSE START RIB MOTION HIGH
IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
ORI     8      ;SET BIT 3 HIGH
OUT     2      ;OUTPUT RESULT
ANI     F7H    ;TURN BIT 3 OFF
OUT     2      ;OUTPUT RESULT

```

If the simulations are combined (F), flip-flops FFC and FFD will switch on at exactly the same instant in time.

The logic in Figure 3-1 shows FFD switching on one clock pulse after FFC. If the clock period is two microseconds, then there will be a two microsecond delay between flip-flops FFD and FFC switching on. Both our simulations are wrong.

Does this matter? We honestly cannot tell with the information at hand. We do not know how external logic uses the FFC and FFD outputs. If the switching time interval between these two flip-flops has to be very close to 2 microseconds, then our simulation is not going to work. Either the two flip-flops must become part of "external logic", or some other means of simulating the eventual overall function must be found.

**TIMING AND
LIMITS OF
SIMULATION**

If external logic demands some switching time delay, but is not fussy about the length of the time delay, then our simulation of flip-flop FFD is adequate.

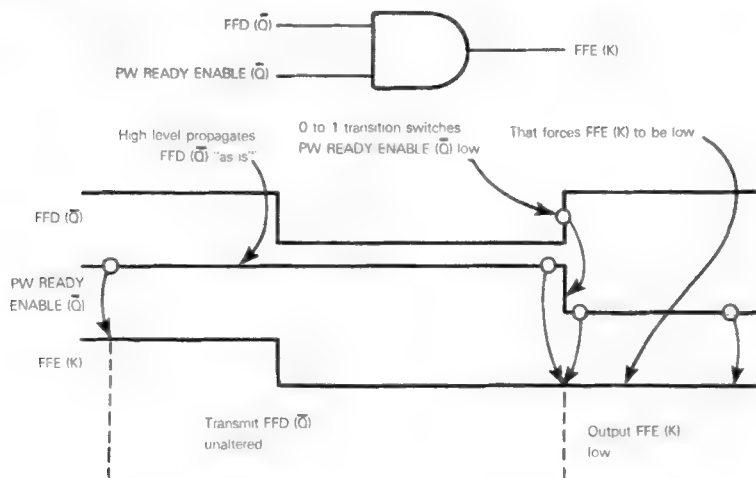
It is quite possible that the logic in Figure 3-1 shows a switching time delay between flip-flops FFC and FFD only to define the leading and trailing edges of the START RIBBON MOTION pulse; but we have taken care of this high pulse by sequentially executing instructions that output 1, then 0 to bit 3 of I/O Port 2. So far as logic internal to Figure 3-1 is concerned, therefore, the need for a switching time delay between flip-flops FFC and FFD disappears. This being the case, we will assume that external logic has no need for a switching time delay between flip-flops FFC and FFD; and we will adopt the shorter, combined simulation identified by **(F)**.

FLIP-FLOP FFE

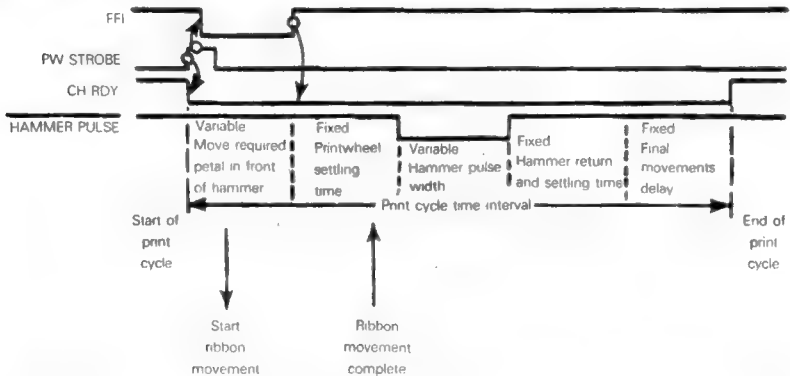
The next device in our logic sequence is flip-flop FFE. The circuitry surrounding this flip-flop is almost identical to FFD.

The FFE (K) input is tied to the FFD (\bar{Q}) output, switched by another component of AND gate 12. The other input to this AND gate is the \bar{Q} output of one-shot 49. One shot 49 is wired in the same way as one-shot 36, which we have just described.

The transition of flip-flop FFD's \bar{Q} output from 0 to 1 will occur when FFD is switched off; and this is the transition which will trigger one-shot 49. Therefore, **one-shot 49 will output \bar{Q} high until flip-flop FFD is switched off, which means that when FFD switches on, its \bar{Q} output will propagate straight through the AND gate connecting it to the FFE (K) input:**



The unique feature of flip-flop FFE is the way in which its J input is generated. This input is the AND of the FFD (Q) output and input signal FFI. Now, the Q output of FFD will go high as soon as FFD switches on, but **FFI is input low from the beginning of the print cycle until the printwheel has correctly positioned itself.** (We described the function of this input signal earlier in the chapter.) **The timing associated with FFI may be illustrated as follows:**



So long as FFI is low, flip-flop FFE will receive a low J input, low J and K inputs, you will recall, hold the Q outputs of a 74107 flip flop in their prior condition. Thus **input signal FFI has been used to create the first time delay of the print cycle: a variable time delay needed to move the required printwheel petal in front of the printhead. Simulating this time delay is simple enough; it may be illustrated as follows:**

:PULSE START RIB MOTION HIGH

```
IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
ORI     8      :SET BIT 3 HIGH
OUT     2      :OUTPUT RESULT
ANI     F7H    :TURN BIT 3 OFF
OUT     2      :OUTPUT RESULT
```

:TEST VELOCITY DECODE INPUT TO CREATE PRINTWHEEL MOVE DELAY

```
VLDC    IN     0      :INPUT I/O PORT 0 TO ACCUMULATOR
RLC     :SHIFT BIT 7 INTO CARRY
JNC     VLDC     :STAY IN LOOP IF CARRY IS 0
```

:AT END OF DELAY SIMULATE FFE SWITCHING ON

```
IN      1      :INPUT I/O PORT 1
ANI     DFH    :RESET BIT 5
ORI     10H    :SET BIT 4
OUT     1      :OUTPUT THE RESULT
```

In order to generate the initial time delay, we simply execute a continuous program loop which inputs the contents of I/O Port 0 to the Accumulator. Bit 7 of I/O Port 0 has been assigned to input signal FFI. We test this bit by shifting it into the Carry status. If the Carry status then has a 0 content,

**TIME DELAY
OF VARIABLE
LENGTH**

FFI must still be low; so we stay within the loop. As soon as a 1 is shifted into the Carry status, the JNC instruction will create a "false" result; the next sequential instruction executes and we are out of the time delay loop:

**JUMP ON
NO CARRY**

VLDC



Jump on Not Carry means jump if Carry is 0 (Not 1). "Jump" means "do not go on to the next sequential instruction", instead go to VLDC

The last four instructions of the FFE simulation show both outputs of this flip-flop becoming output signals. This meets requirements of Figure 3-1. We therefore reset bit 5 (it represents the \bar{Q} output) and we set bit 4 (it represents the Q input).

The instruction sequence executed in between print cycles will have to be modified to ensure that bit 5 has initially been set to 1, while bit 4 has initially been reset to 0. Here are the required modifications:

:IN BETWEEN PRINT CYCLES PROGRAM EXECUTION

:INITIALLY SET I/O PORT BITS 4, 3, 2 AND 1 TO 0, BITS 5 AND 0 TO 1

```

IN      1      :INPUT I/O PORT 1 TO ACCUMULATOR
ORI     21H    :SET BITS 5 AND 0 TO 1
ANI     E1H    :RESET BITS 4, 3, 2 AND 1 TO 0
OUT     1      :RETURN RESULT

```

:TEST FOR RETURN STROBE LOW

```

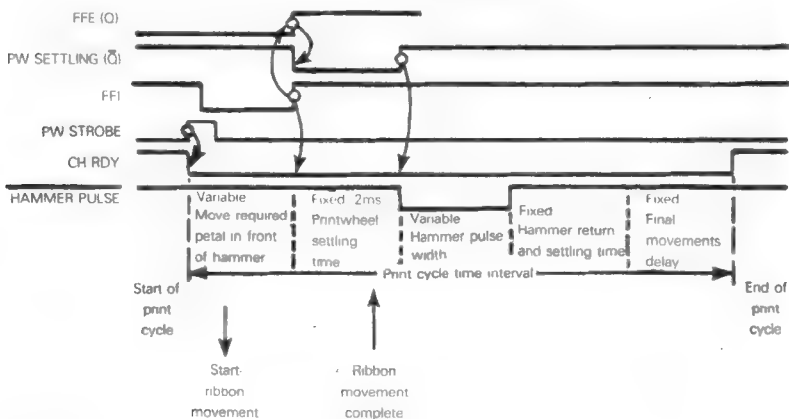
L10     IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     10H    :ISOLATE RETURN STROBE
        JZ      FFB    :IF IT IS 0, JUMP TO FFB SIMULATION

```

PW SETTLING ONE-SHOT

The PW SETTLING one-shot is the 74121 device at co-ordinate B5 in Figure 3-1. We have described this device in Chapter 2. With its two A inputs tied to ground, this one-shot is triggered by a low-to-high transition at its B input. Since the B input is tied to the FFE Q output, this transition occurs as soon as flip-flop FFE switches on.

The PW SETTLING one-shot has a two millisecond delay. This delay results from the external capacitor/resistor combination marked C1 and R1. Therefore as soon as FFE switches on; the PW SETTLING one-shot outputs \bar{Q} low for two-milliseconds:



SIMULATING THE PW SETTLING ONE-SHOT

Simulating the one-shot time delay is simple enough and may be illustrated as follows:

ONE-SHOT
TIME DELAY
SIMULATION

```

:PULSE START RIB MOTION HIGH
      IN      2      :INPUT I/O PORT 2 TO ACCUMULA-
                        TOR
      ORI      8      :SET BIT 3 HIGH
      OUT      2      :OUTPUT RESULT
      ANI      F7H     :TURN BIT 3 OFF
      OUT      2      :OUTPUT RESULT
:TEST VELOCITY DECODE INPUT TO CREATE PRINTWHEEL MOVE DELAY
VLDC   IN      0      :INPUT I/O PORT 0 TO ACCUMULA
                        TOR
      RLC      :SHIFT BIT 7 INTO CARRY
      JNC      VLDC    :STAY IN LOOP IF CARRY IS 0
:AT END OF DELAY SIMULATE FFE SWITCHING ON
      IN      1      :INPUT I/O PORT 1
      ANI      DFH     :RESET BIT 5
      ORI      10H     :SET BIT 4
      OUT      1      :OUTPUT THE RESULT
SIMULATE 2 MS PW SETTLING TIME DELAY
      MVI      A,0      :LOAD ACCUMULATOR WITH 0
PWS    DCR      A        :DECREMENT A
      JNZ      PWS      :IF A DOES NOT DECREMENT TO 0,
                        RE-DECREMENT
    
```

There are some interesting variations in the PW SETTLING, two millisecond time delay loop illustrated.

The time delay loop consists of just two instructions: the DCR A instruction decrements the contents of the Accumulator and the JNZ PWS instruction returns to DCR A if the Accumulator does not contain zero after the decrement is complete. These two instructions execute in 15 clock periods, which add up to 7.5 microseconds using a 500 nanosecond clock.

By initially loading 0 into the Accumulator, these two instructions will be executed 256 times, since the first decrement of the Accumulator will take it from 0 to FF₁₆. Thus the total time delay is given by the following equation:

$$256 \times 7.5 + 3.5 = 1923.5 \text{ microseconds}$$

Initial Accumulator contents

Time to execute DCR and JNZ instructions once

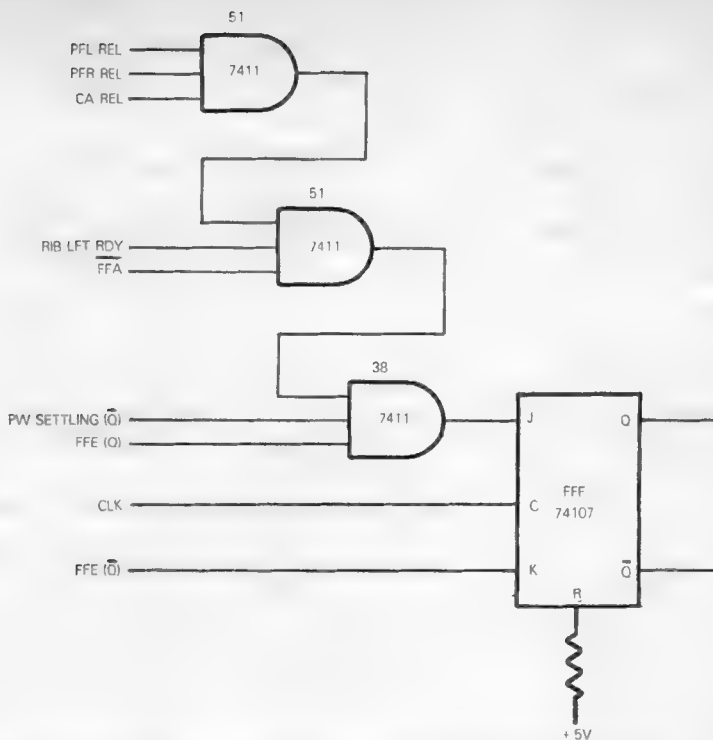
Time to execute MVI A,0 instruction

1923.5 microseconds equals 0.19235 milliseconds.

This is close enough to 2 milliseconds for our purposes.

FLIP-FLOP FFF

Once the PW settling one-shot has timed out, we are ready to fire the printhead. The 555 multivibrator is actually going to generate the printhead firing pulse, but it is most important to ensure that the printhead does not fire while any part of the print or carriage mechanisms are moving. The 555 one-shot is therefore triggered by flip-flop FFF which, in turn, is switched on by a J input that is the AND of many safeguard signals. Let us isolate flip-flop FFF and examine its inputs.



With its Clear (R) input tied to +5V, flip-flop FFF has the following function table:

Inputs		Outputs	
J	K	Q	\bar{Q}
0	0	No change	
1	0	1	0
0	1	0	1
1	1	Complement	

Inputs at positive clock edge

Control outputs at negative clock edge

CLOCK

In between print cycles, FFE is "off", so the K input to FFF is high. The flip-flop FFF J input will be low since the FFE (Q) output will be low, and FFE (Q) is one contributor to FFF (J).

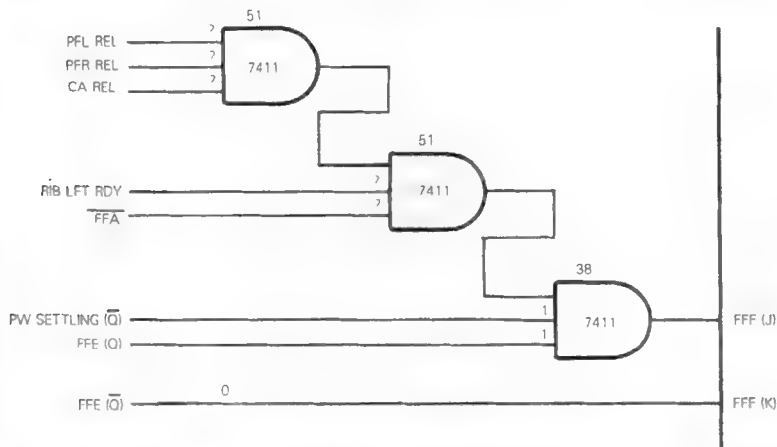
In between print cycles, therefore, flip-flop FFF is "off", since a low J input and a high K input generate steady outputs of $Q = 0$, $\bar{Q} = 1$; this is characteristic of a flip-flop in its "off" condition.

Now when FFE switches on, it inputs a low K to FFF. So long as the J input is also low, no change occurs. As soon as the seven signals contributing to FFF (J) are all high, flip-flop FFF will receive a high J input; this will switch flip-flop FFF on — Q is then output high and \bar{Q} is output low.

SIMULATING FLIP-FLOP FFF

Coming out of the simulation of FFE, we know that FFE (Q) and FFE (\bar{Q}) have correct levels for FFF to switch on.

Coming out of the simulation of the PW SETTLING one-shot, the one-shot \bar{Q} output must be high:



All that is needed is to test the five remaining interlock signals; as soon as they are all high, we simulate flip-flop FFF switching on. This is the instruction sequence:

;PULSE START RIB MOTION HIGH

```
IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
ORI     8      ;SET BIT 3 HIGH
OUT     2      ;OUTPUT RESULT
ANI     F7H    ;TURN BIT 3 OFF
OUT     2      ;OUTPUT RESULT
```

;TEST VELOCITY DECODE INPUT TO CREATE PRINTWHEEL MOVE DELAY

```
VLDC    IN     0      ;INPUT I/O PORT 0 TO ACCUMULATOR
RLC     ;SHIFT BIT 7 INTO CARRY
JNC     VLDC    ;STAY IN LOOP IF CARRY IS 0
```

;AT END OF DELAY SIMULATE FFE SWITCHING ON

```
IN      1      ;INPUT I/O PORT 1
ANI     DFH    ;RESET BIT 5
ORI     10H    ;SET BIT 4
OUT     1      ;OUTPUT THE RESULT
```

;SIMULATE 2 MS PW SETTLING TIME DELAY

```
MVI     A,0    ;LOAD ACCUMULATOR WITH 0
PWS     DCR     A ;DECREMENT A
JNZ     PWS     ;IF A DOES NOT DECREMENT TO 0,
                ;RE-DECREMENT
```

```

;SIMULATE FLIP-FLOP FFF SWITCHING ON
FFF      IN      0      ;INPUT I/O PORT 0 CONTENTS TO ACCUMULATOR
        CMA      ;COMPLEMENT TO TEST FOR 1 BITS
        ANI      1FH    ;ISOLATE BITS 0 THROUGH 4
        JNZ      FFF    ;IF THERE WERE ANY 0 BITS, STAY IN LOOP
        IN      1      ;INPUT I/O PORT 1 TO ACCUMULATOR
        ORI      40H    ;SET BIT 6 TO 1
        OUT      1      ;OUTPUT THE RESULT

```

By now, you should be able to understand instructions as they are added to the program.

The first four instructions simply load in the contents of I/O Port 0 and test for 1s in the low order five bits. Until such time as all five bits are 1, the program will remain in the four-instruction loop that begins with IN 0 and ends with JNZ FFF.

When bits 0 through 4 all equal 1, the CMA instruction changes all these bits to 0:

			Accumulator Contents
FFF	IN	0	XXX11111
	CMA		XXX00000
	ANI	1FH	00011111
			00000000 Zero status = 1
	JNZ	FFF	Return to FFF only if Zero status = 0
	IN	1	Continue here if Zero status is 1

The JNZ instruction no longer deflects program execution back to FFF, rather, it allows the next sequential instruction to be executed.

The last three instructions simulate flip-flop FFF being switched on. Bit 6 of I/O Port 1 has been assigned to FFF, therefore, this is the bit which has to be set to 1.

We can make the final modification to the instruction sequence which correctly sets flip-flop status in between print cycles. This is what we finish up with:

```

;IN BETWEEN PRINT CYCLES PROGRAM EXECUTION
;INITIALLY SET I/O PORT BITS 6, 4, 3, 2 AND 1 TO 0, 5 AND 0 TO 1
        IN      1      ;INPUT I/O PORT 1 TO ACCUMULATOR
        ORI      21H    ;SET BITS 5 AND 0 TO 1
        ANI      A1H    ;RESET BITS 6, 4, 3, 2 AND 1 TO 0
        OUT      1      ;RETURN RESULTS
;TEST FOR RETURN STROBE LOW
L10      IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI      10H    ;ISOLATE RETURN STROBE
        JZ       FFB    ;IF IT IS 0, JUMP TO FFB SIMULATION

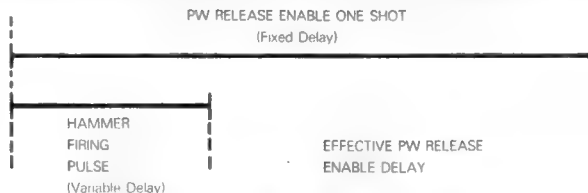
```

What happens when flip-flop FFF switches on?

The FFF (Q) output goes up to pin 9 of AND gate 37 at coordinate A6. This is part of the logic which contributes to the PW REL signal. However, the transition of the FFF (Q) output from low-to-high is not significant, since the other input to AND gate 37 is the FFD (Q) output which is currently low. The FFF (Q) output is connected to AND gate 37 to hold PW REL low early in the print cycle when FFD (Q) is high.

The FFF Q and \bar{Q} outputs contribute to the FFC J and K inputs. FFF (\bar{Q}) is one contributor to AND gate 12, the output of which becomes the FFC (J) input. The other contributor to this AND gate is the output of AND gate 37 at coordinate A3, which is constantly high by this time in the print cycle, therefore, when the FFF (\bar{Q}) output goes low the FFC (J) input also goes low. The K input to FFC is the FFF (Q) output. **FFC will therefore switch off when K goes high and that will not happen until FFF switches on.**

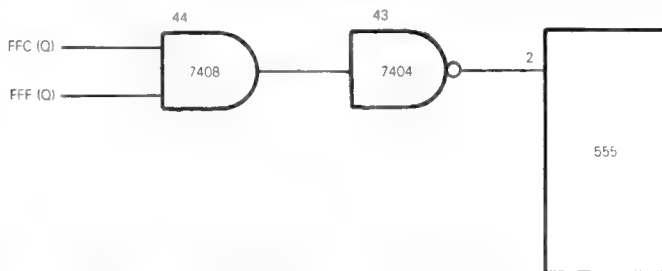
In our simulation, however, we are going to postpone FFC switching off until the end of HAMMER PULSE. This is because the purpose of FFC switching off is to trigger the PW RELEASE ENABLE one-shot, which creates the time delay needed by the printhead to settle back. Thus instead of using parallel delays:



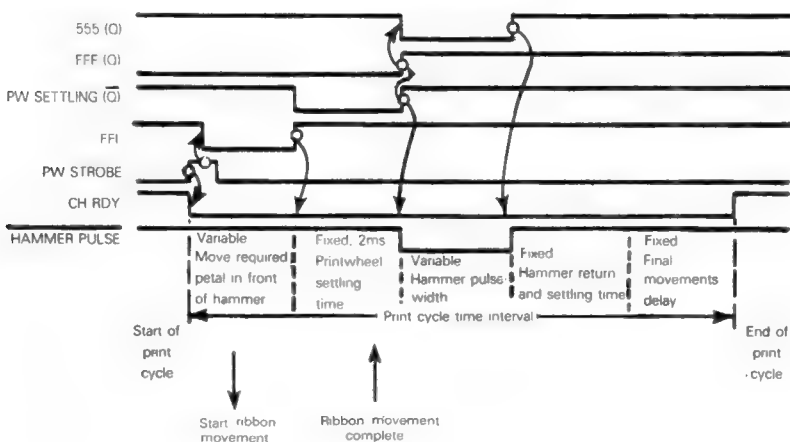
we will implement serial delays, which more immediately meet logic needs:



The hammer firing pulse is generated by the 555 one-shot. Therefore the 555 one-shot provides the next event in our chronological sequence; it is triggered by a low-to-high transition at pin 2. This pin input is created as follows:



This is the sequence of events that must be simulated:



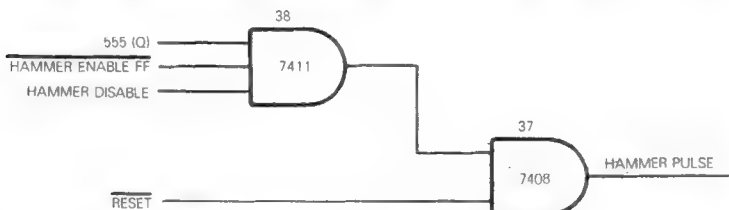
THE 555 MULTIVIBRATOR

Compare the way in which the 555 multivibrator has been wired in Figure 3-1 with the description of the multivibrator, as given in Chapter 2; you will see that **flip-flop FFB switches the multivibrator "off" in between print cycles** by inputting a low reset at pin 4. **The flip-flop FFF (Q) output triggers the multivibrator**, as we have just described.

The duration of the one-shot output pulse is controlled by inputs H1 through H6. One of these six inputs will be true while the other five will be false; thus the multivibrator, once triggered, will output a one-shot which can have a "high" pulse with one of six possible durations.

**ONE-SHOT
VARIABLE
PULSE**

The 555 multivibrator one-shot output is eventually inverted to become a hammer pulse output; however, for the hammer pulse output to occur, additional inputs to AND gates 37 and 38, located at co-ordinates B8 and C7, respectively, must also be high. We may represent the hammer pulse logic as follows:



We will simply have to test the HAMMER ENABLE FF input before generating a HAMMER PULSE output.

The HAMMER DISABLE switch must be simulated.

RESET we can ignore, since RESET logic is being simulated in between print cycles.

SIMULATING MULTIVIBRATOR 555

The simulation of the 555 multivibrator consists of the following logic sequence:

- 1) **Determine if conditions have been satisfied for a 555 one-shot output to be transmitted as a HAMMER PULSE output.**
- 2) **Examine inputs H1 through H6. Based on these inputs, create one of six possible time delays.**
- 3) **If conditions for a HAMMER PULSE output have been satisfied, translate the 555 one-shot output into a HAMMER PULSE output.**

Let us first look at the HAMMER PULSE output enabling logic. Testing the condition of HAMMER ENABLE FF is simple enough, it has been assigned pin 6 of I/O Port 0.

But there are no switches in assembly language programs; how are we going to simulate the hammer disable? We could assign the one remaining pin — pin 5 of I/O Port 0 to an input signal generated by an external switch. It would be just as simple to place this switch in the path of HAMMER ENABLE FF as follows:

**LOGIC EXCLUDED
FROM
MICROCOMPUTER**



We will therefore ignore the hammer disable switch and enable a hammer pulse output providing the HAMMER ENABLE FF input is high.

What about the six possible durations for the 555 multivibrator output? We described in Chapter 2 how a time delay can be created by loading a 16-bit value into two registers, then decrementing those registers within a program loop, remaining in the program loop until a decrement to zero occurs. Here the instruction loop is repeated:

```

                LXI      D,T16      ;LOAD TIME CONSTANT INTO D AND E
LOOP           DCX      D           ;DECREMENT DE
                MOV      A,D        ;TEST FOR ZERO BY ORING
                OR       E          ;D AND E CONTENTS VIA ACCUMULATOR
                JNZ      LOOP

```

Selecting one of six possible time delays is as simple as selecting one of six possible initial time constants. We can now simulate our 555 multivibrator as follows:

```

                IN       1          ;INPUT I/O PORT 1 TO ACCUMULATOR
                ORI      40H        ;SET BIT 6 TO 1
                OUT      1          ;OUTPUT THE RESULT
;TEST HAMMER ENABLE FF
                IN       0          ;INPUT I/O PORT 0 TO ACCUMULATOR
                ANI      40H        ;ISOLATE BIT 6
                JZ       HPO        ;IF ZERO BYPASS SETTING HAMMER PULSE LOW
;HAMMER ENABLE FF IS HIGH SO HAMMER PULSE
;MUST BE OUTPUT LOW THEREFORE SET BIT 2 OF
;/O PORT 3 TO 0
                IN       2          ;INPUT I/O PORT 2 TO ACCUMULATOR
                ANI      FBH        ;SET BIT 2 TO 0
                OUT      2          ;OUTPUT RESULT
;COMPUTE TIME DELAY
HPO            LXI      H,DELY      ;LOAD DATA ADDRESS BASE INTO HL
                LDA      H1H6       ;LOAD SELECTOR INTO ACCUMULATOR
HP1            RRC              ;ROTATE ACCUMULATOR, SET CARRY TO A0
                INX      H          ;INCREMENT HL CONTENTS BY 2
                INX      H
                JNC      HP1        ;IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
                MOV      D,M        ;LOAD 16-BIT TIME DELAY CONSTANT INTO DE
                INX      H
                MOV      E,M
TDLY           DCX      D           ;EXECUTE TIME DELAY LOOP
                MOV      A,D
                ORA      E
                JNZ      TDLY
;OUTPUT HAMMER PULSE HIGH AGAIN
                IN       2          ;INPUT I/O PORT 2 TO ACCUMULATOR
                ORI      4          ;SET BIT 2 TO 1
                OUT      2          ;OUTPUT RESULT

```

Compared to the other devices we have simulated thus far, the 555 multivibrator requires a lot of simulation instructions. While it may look as though there is a lot to understand, the logic is, in fact, quite simple; so let us take it one piece at a time.

Initially we test HAMMER ENABLE FF. HAMMER PULSE will be output low only if HAMMER ENABLE FF is high. The three instructions which test the status of HAMMER ENABLE FF are:

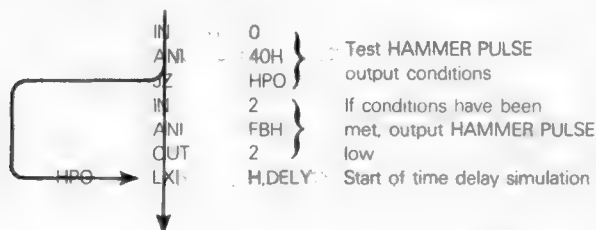
```

                IN       0          ;INPUT I/O PORT 0 TO ACCUMULATOR
                ANI      40H        ;ISOLATE BIT 6
                JZ       HPO        ;IF ZERO, BYPASS SETTING HAMMER PULSE LOW

```

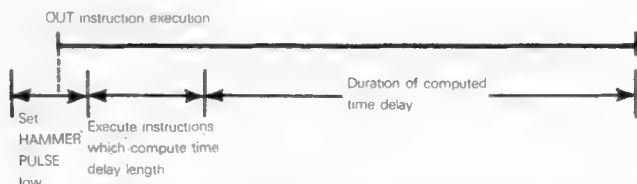
**SIGNAL
ENABLE**

There are two aspects of these three instructions which need to be explained. First, there is the logic being implemented. We are determining if conditions have been met for HAMMER PULSE to be output low. If conditions have been met, then HAMMER PULSE will be output low immediately; if conditions have not been met, the JZ HPO instruction branches around the instruction sequence that outputs HAMMER PULSE low:

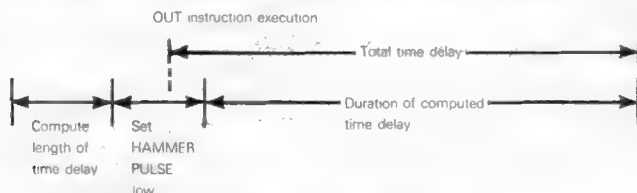


We output hammer pulse low before starting to compute the duration of the time pulse; why is this? The reason is to save time. Instructions which compute the length of the time delay can be executed at the beginning of the time delay:

EVENT SEQUENCE



We could just as easily have computed the time delay, then set HAMMER PULSE low, then executed the time delay; events would have occurred chronologically as follows:



Overlapping events in time makes a lot more sense.

The actual method used to compute the time delay needs a little explanation. **At the end of our program, there will be 12 bytes of memory in which six 16-bit constants are stored.** This is how the source program will look:

```

HPO  LXI  H, DELAY  ;LOAD ADDRESS OF FIRST DELAY INTO HL
     LDA  H1H6      ;LOAD SELECTOR INTO ACCUMULATOR
HP1  RRC           ;ROTATE ACCUMULATOR, SET CARRY TO A0
     INX  H         ;INCREMENT HL CONTENTS BY 2
     INX  H
     JNC  HP1       ;IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
     MOV  D, M      ;LOAD 16-BIT TIME DELAY CONSTANT INTO DE
     INX  H
     MOV  E, M
  
```

```

TDLY    DCX    D    ;EXECUTE TIME DELAY LOOP
        MOV    A,D
        ORA    E
        JNZ    TDLY
;OUTPUT HAMMER PULSE HIGH AGAIN
        IN     2    ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI    4    ;SET BIT 2 TO 1
        OUT    2    ;OUTPUT RESULT

```

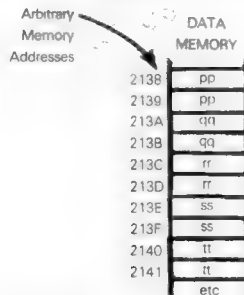
```

ORG      DELY + 2
pppp      ;H1 TIME DELAY
qqqq      ;H2 TIME DELAY
rrrr      ;H3 TIME DELAY
ssss      ;H4 TIME DELAY
tttt      ;H5 TIME DELAY
uuuu      ;H6 TIME DELAY

```

The letters p, q, r, s, t and u have been used to represent hexadecimal values. The six time delays can be represented by any numeric values, ranging from 0000_{16} through $FFFF_{16}$.

The address of the first memory byte in which the first time delay is stored is given by the expression $DELY + 2$. Suppose this memory location happened to be 2138:



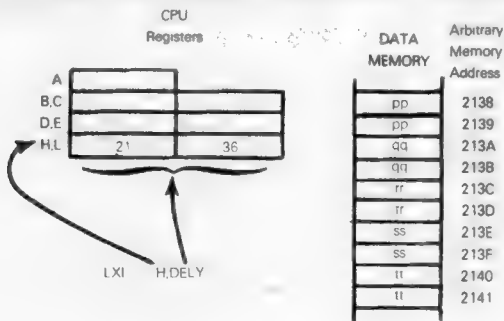
DELY is a label to which the value 2136 must be assigned. This assignment is made using an Equate directive, which would appear at the beginning of the program as follows:

```

DELY    EQU    1316H

```

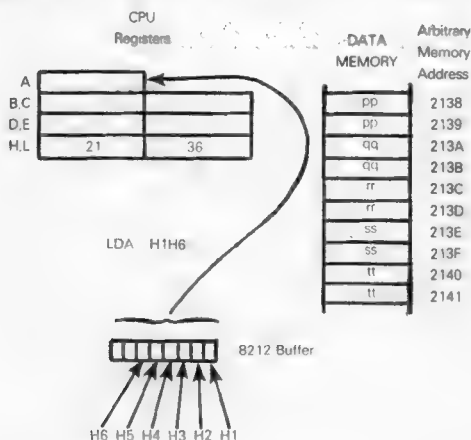
Now we begin our computation of the time delay by loading the address DELY into the H and L registers. Assume that the label DELY has the value 2316H, as illustrated above. After the LXI H,DELY instruction has been executed this is the situation:



The next instruction, LDA H1H6, loads the contents of the 8212 8-bit buffer into the Accumulator. The memory address which causes the buffer to select itself is represented by the label H1H6. Suppose this memory address is FFFF₁₆; then H1H6 would have to be assigned the value FFFF₁₆ using an Equate directive at the beginning of the program, as follows.

```
DELY EQU 2316H
H1H6 EQU FFFFH
```

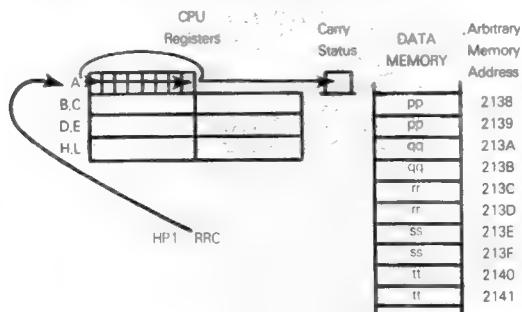
From our discussion of input signals, recall that of the six inputs H1 through H6, one signal will be high while the other five signals are low. Therefore, **after the LDA instruction has executed, the Accumulator will contain a 1 in one of the six low order bits:**



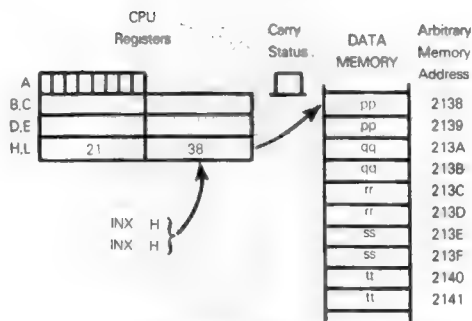
We can compute the address of the required time delay by adding 2 to the contents of the H and L registers a number of times given by the position of the Accumulator 1 bit. This may be illustrated as follows:

DATA MEMORY ADDRESS COMPUTATION

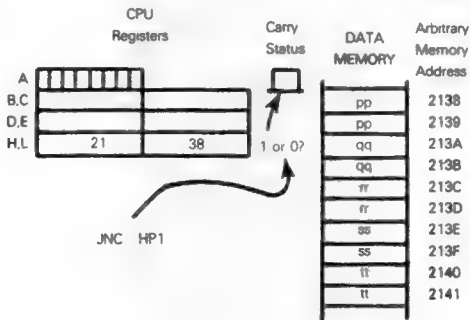
- ① Shift Accumulator contents right one bit, and into carry:



- ② Add 2 to HL:



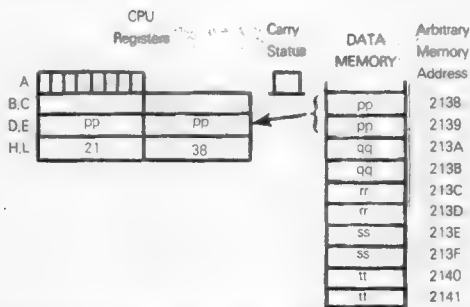
- ③ If Carry status is not 1, go back to 1; otherwise HL contains the correct address:



The logic to make the required address addition is provided by these four instructions

```
HP1  RRC      ;ROTATE ACCUMULATOR, SET CARRY TO A0
      INX     H      ;INCREMENT HL BY 2
      INX     H
      JNC     HP1    ;IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
```

Now that the correct time delay is addressed by the H and L registers, we load the appropriate 16-bit delay constant into D and E. Suppose H1 was the high input signal; this is the result:



The selected address pppp is moved to the D and E registers by the three instructions:

```
MOV     D,M    ;MOVE CONTENTS OF BYTE 2138 TO D REGISTER
INX     H      ;ADDRESS BYTE 2139
MOV     E,M    ;MOVE CONTENTS OF BYTE 2139 TO E REGISTER
```

The actual time delay is created by this instruction loop, which was described in Chapter 2:

```
TDLY    DCX     D    ;DECREMENT DELAY COUNTER
        MOV     A,D   ;TEST FOR 0 IN DE BY ORING D
        ORA     E     ;WITH E IN ACCUMULATOR
        JNZ     TDLY  ;RETURN IF NOT ZERO
```

The last three instructions output HAMMER PULSE high, without making any test for whether HAMMER PULSE was low. This logic will work since outputting HAMMER PULSE high, if it was already high, will have no discernable effect. Under these circumstances, the time required to execute the last three instructions is simply wasted. Since it would take three instructions to test if HAMMER PULSE had been set low, the waste is justified.

Let us now give a little thought to the time it will take to compute the time delay. Execution times for relevant instructions are listed as follows:

TIME DELAY COMPUTATION

Cycles		Instruction	
		IN 2	
		ANI FBH	
		OUT 2	
10	HP0	LXI H,DELY	← Hammer pulse low starts here
13		LDA H1H6	
4	HP1	RRC	} These four instructions will be executed between 1 and 6 times. 26 cycles are in this loop.
5		INX H	
5		INX H	
10		JNC HP1	
7		MOV D,M	
5		INX H	
7		MOV E,M	
5	TDLY	DCX D	
5		MOV A,D	
4		ORA E	
10		JNZ TDLY	
10		IN 2	
7		ORI 4	
10		OUT 2	← Hammer pulse low ends here
117			

Assuming a 500 nanosecond clock, execution time is given by the expression:

$$(46.5 + 12 \times N) \text{ microseconds}$$

where N is a number between 1 for the shortest pulse, and 6 for the longest pulse. Therefore **execution times will range between 58.5 microseconds and 118.5 microseconds. These times must be subtracted from the delays subsequently generated.** For example, suppose H1 high requires the 555 to output a one shot signal which is high for 1.65 milliseconds (approximately); then a delay of 16 milliseconds, added to a set up time of 58.5 microseconds will suffice.

THE PW RELEASE ENABLE FLIP-FLOP

As soon as the 555 one-shot output becomes low again, flip-flop FFC is simulated switching off. When FFC switches off, its \bar{Q} output makes a low-to-high transition and this triggers the PW RELEASE ENABLE one-shot. This is a 74121 one-shot, identified by the 36 at approximately co-ordinate E2. The purpose of this one shot is to allow the printhead time to settle back before any attempt is made to reposition the printwheel. This was illustrated as the fixed, hammer return and settling time delay.

SIMULATING THE PW RELEASE ENABLE FLIP-FLOP

This is really a two-part simulation; first we must simulate flip-flop FFC switching off, then we must execute an appropriate time delay. A 3 millisecond time delay is sufficient.

TIME DELAY

Instructions which turn flip-flop FFC off will execute within the 3 millisecond time delay. The computed time delay will therefore be a little less than 3 milliseconds. Here is the appropriate instruction sequence:

JNC	HP1	:IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
MOV	D,M	:LOAD 16-BIT TIME DELAY CONSTANT INTO DE
INX	H	
MOV	E,M	

```

TDLY    DCX    D        ;EXECUTE TIME DELAY LOOP
        MOV    A,D
        ORA    E
        JNZ    TDLY
;OUTPUT HAMMER PULSE HIGH AGAIN
        IN     2        ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI    4        ;SET BIT 2 TO 1
        OUT    2        ;OUTPUT RESULT
;SWITCH FLIP-FLOP FFC OFF
        IN     1        ;INPUT I/O PORT 1 TO ACCUMULATOR
        ANI    FBH      ;SET BIT 2 TO 0
        OUT    1        ;OUTPUT RESULTS
;EXECUTE A 3 MILLISECOND TIME DELAY
        LXI    D,F7H    ;LOAD TIME CONSTANT INTO D,E
PWR1     DCX    D        ;DECREMENT D,E
        MOV    A,D      ;TEST FOR ZERO RESULT
        ORA    E
        JNZ    PWR1     ;REDECREMENT IF NOT ZERO

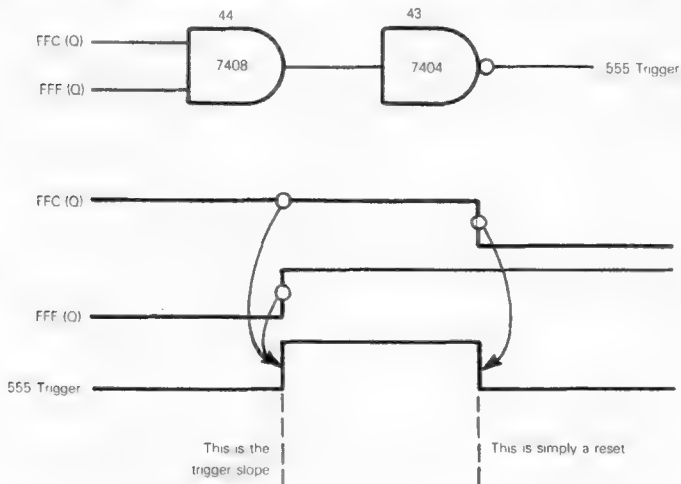
```

The time constant F7H is equal to 247_{10} . The four preceding instructions execute in 34 microseconds, and the four instructions in the delay loop execute in 12 microseconds. Therefore the total delay time is given by the equation:

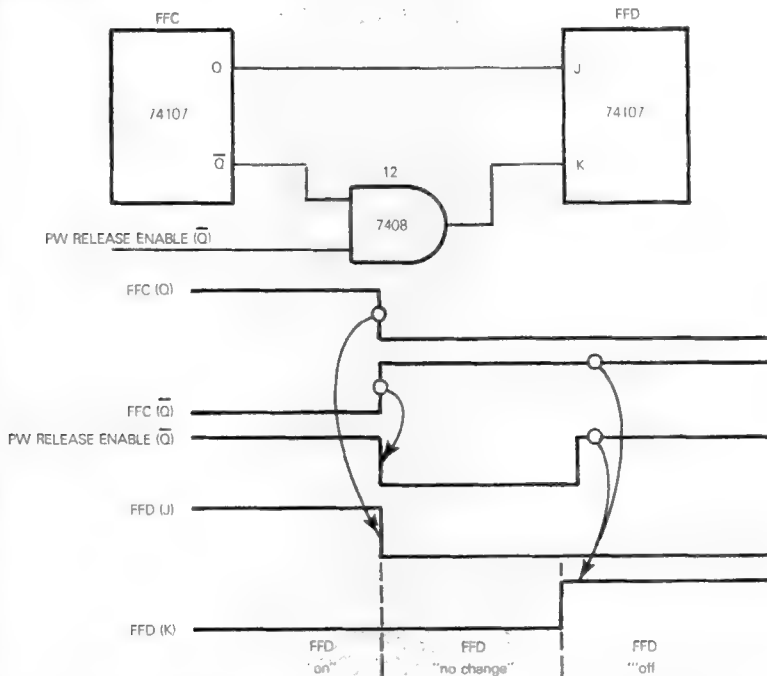
$$247 \times 12 + 34 = 2998 \text{ microseconds}$$

To be honest, the 3 millisecond time delay is not a critical number; 2.5 or 3.5 milliseconds would probably do just as well, so our worrying about 34 microseconds is not meaningful in this instance. Nevertheless, in your next application, the duration of a time delay may be very critical, then the timing considerations discussed above will be very meaningful.

In order to determine what happens at the conclusion of the PW RELEASE time delay, we must look at the FFC Q and \bar{Q} outputs. The Q output connects to the START RIBBON MOTION PULSE AND gate, and to the 555 one-shot trigger logic; in neither case does the Q high-to-low transition have any effect. The START RIBBON MOTION pulse signal is already low and the 555 one-shot is triggered by a low-to-high Q transition. The high-to-low transition simply drops the trigger signal to a low level which requires no simulation:



The FFC (\bar{Q}) output is ANDed with the PW RELEASE ENABLE \bar{Q} one-shot in order to generate the FFD (K) input. The FFD (J) input comes directly from FFC (Q), therefore **as soon as the PW RELEASE ENABLE one-shot goes high again, FFD will receive a low J input and a high K input:**



A low J and high K input to flip-flop FFD switches this flip-flop off; and that triggers the PW READY ENABLE one-shot.

SIMULATING THE PW READY ENABLE ONE-SHOT

Logic associated with this one-shot is almost identical to the PW RELEASE ENABLE one-shot. FFD switching off causes a low to-high \bar{Q} output, which triggers the PW READY ENABLE one-shot.

We must now simulate a 2 millisecond time delay; otherwise the next instruction sequence is almost identical to the PW RELEASE ENABLE one-shot simulation and may be illustrated as follows:

	JNC	HP1	:IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
	MOV	D,M	:LOAD 16-BIT TIME DELAY CONSTANT INTO DE
	INX	H	
	MOV	E,M	
TDLY	DCX	D	:EXECUTE TIME DELAY LOOP
	MOV	A,D	
	ORA	E	
	JNZ	TDLY	


```

:OUTPUT HAMMER PULSE HIGH AGAIN
    IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
    ORI     4      :SET BIT 2 TO 1
    OUT     2      :OUTPUT RESULT

:SWITCH FLIP-FLOP FFC OFF
    IN      1      :INPUT I/O PORT 1 TO ACCUMULATOR
    ANI     FBH    :SET BIT 2 TO 0
    OUT     1      :OUTPUT RESULTS

:EXECUTE A 3 MILLISECOND TIME DELAY
    LXI     D,F7H  :LOAD TIME CONSTANT INTO D,E
PWR1      DCX     D      :DECREMENT D,E
    MOV     A,D    :TEST FOR ZERO RESULT
    ORA     E
    JNZ     PWR1   :REDECREMENT IF NOT ZERO

:SWITCH FLIP FLOP FFD OFF
    IN      1      :INPUT I/O PORT 1 TO ACCUMULATOR
    ANI     F7H    :SET BIT 3 TO 0
    OUT     1      :OUTPUT RESULTS

:EXECUTE A 2 MILLISECOND TIME DELAY
    MVI     A,83H  :LOAD TIME CONSTANT INTO ACCUMULATOR
PWR2      DCR     A      :DECREMENT ACCUMULATOR
    JNZ     PWR2   :REDECREMENT IF NOT ZERO

```

When FFD switches off, the PW REL output goes high again. Here is the PW REL creation logic:



FFB (\bar{Q}) is still low at this time. But FFD (\bar{Q}) and FFF (\bar{Q}) are both high so AND gate 37 outputs a high level which passes through OR gate 26 to set PW REL high.

These instructions set PW REL high:

```

    JNC     HP1    :IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
    MOV     D,M    :LOAD 16-BIT TIME DELAY CONSTANT INTO DE
    INX     H
    MOV     E,M
TDLY      DCX     D      :EXECUTE TIME DELAY LOOP
    MOV     A,D
    ORA     E
    JNZ     TDLY

:OUTPUT HAMMER PULSE HIGH AGAIN
    IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
    ORI     4      :SET BIT 2 TO 1
    OUT     2      :OUTPUT RESULT

:SWITCH FLIP-FLOP FFC OFF
    IN      1      :INPUT I/O PORT 1 TO ACCUMULATOR
    ANI     FBH    :SET BIT 2 TO 0
    OUT     1      :OUTPUT RESULTS

```

```

:EXECUTE A 3 MILLISECOND TIME DELAY
PWR1  LXI    D,F7H    ;LOAD TIME CONSTANT INTO D,E
      DCX    D        ;DECREMENT D,E
      MOV    A,D      ;TEST FOR ZERO RESULT
      ORA    E
      JNZ    PWR1     ;REDECREMENT IF NOT ZERO
:SWITCH FLIP-FLOP FFD OFF
      IN     1        ;INPUT I/O PORT TO ACCUMULATOR
      ANI    F7H      ;SET BIT 3 TO 0
      OUT    1        ;OUTPUT RESULT
:EXECUTE A 2 MILLISECOND TIME DELAY
      MVI    A,83H    ;LOAD TIME CONSTANT INTO ACCUMULATOR
PWR2   DCR    A        ;DECREMENT ACCUMULATOR
      JNZ    PWR2     ;REDECREMENT IF NOT ZERO
:SET PW REL HIGH
      IN     2        ;INPUT I/O PORT 2 TO ACCUMULATOR
      ORI    1        ;SET BIT 0 TO 1
      OUT    2        ;OUTPUT RESULT

```

Now the whole print cycle ends in a hurry. The flip-flop FFD Q and \bar{Q} outputs become the FFE J and K inputs. Q is first ANDed with FFI which, at this time, is constantly high; therefore the moment FFD switches off, FFE receives a low J input.

The FFE (K) input does not go high until the end of the PW READY ENABLE one-shot, since the PW READY ENABLE \bar{Q} output is ANDed with \bar{Q} from FFD in order to generate FFE (K).

FFE switching off is our next chronological event.

FFE switching off, in turn, causes FFB and FFF to switch off. FFB is switched off by the low-to-high transition of FFE (\bar{Q}) which becomes the FFB clock input. FFF switches off because its J and K inputs are tied directly to the Q and \bar{Q} outputs of FFE.

Once FFB and FFF have switched off, all conditions have been met for CH RDY to go high again, providing EOR DET is not signaling the end of ribbon:



Thus we may conclude our simulation as follows:

```

        JNZ     PWR1      ;REDECREMENT IF NOT ZERO
;SWITCH FLIP-FLOP FFD OFF
        IN      1         ;INPUT I/O PORT 1 TO ACCUMULATOR
        ANI     F7H       ;SET BIT 3 TO 0
        OUT     1         ;OUTPUT RESULT
;EXECUTE A 2 MILLISECOND TIME DELAY
        MVI     A,83H     ;LOAD TIME CONSTANT INTO ACCUMULATOR
PWR2    DCR     A         ;DECREMENT ACCUMULATOR
        JNZ     PWR2      ;REDECREMENT IF NOT ZERO
;SET PW REL HIGH
        IN      2         ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     1         ;SET BIT 0 TO 1
        OUT     2         ;OUTPUT RESULT
;TURN OFF FLIP-FLOPS FFB, FFE AND FFF
        IN      1         ;INPUT I/O PORT 1 TO ACCUMULATOR
        ANI     ADH       ;RESET BITS 1, 4 AND 6 TO 0
        ORI     20H       ;SET BIT 5 TO 1
        OUT     1         ;OUTPUT RESULTS
;SET CH RDY HIGH
        IN      2         ;INPUT I/O PORT 2 TO ACCUMULATOR
        ORI     2         ;SET BIT 1 TO 1
        OUT     2         ;OUTPUT RESULT
;BRANCH TO TEST FOR VALID END OF PRINT CYCLE
        JMP     LOP1

```

SIMULATION SUMMARY

The complete simulation program developed in this chapter is given in Figure 3-3.

We can conclude that an absolutely exact, one-for-one simulation of digital logic using assembly language instructions within a microcomputer system is not feasible; but then it is not particularly desirable.

If you are not a digital logic designer, you will probably be very confused by the various signal combinations required within the logic of Figure 3-1. A great deal of what is going on has nothing to do with the ultimate requirements of the Qume printer; rather, it reflects one logic designer's internal logic implementation, aimed at insuring appropriate external signal sequences under all conceivable circumstances.

If you are a logic designer, chances are you would have implemented the specific requirements of the Qume printer interface in a totally different way; you may even be grumbling at the implementation.

The important point to bear in mind is that digital logic contains innumerable subtleties which are specific to discrete logic devices. These subtleties are not tied to the requirements of the overall implementation.

Now assembly language has its own set of subtleties, which also have nothing to do with the ultimate implementation; rather, they are aimed at making most effective use of individual instructions or instruction sequences.

It should therefore come as no surprise that an exact duplication of digital logic, using assembly language, is neither feasible nor desirable. So we will move away from digital logic and start treating a problem from a programming viewpoint.

The principle difference between digital logic and assembly language is that assembly language treats events chronologically, while digital logic segregates logic into functional nodes. Thus, one logic device may be responsible for a number of events occurring at different times during any logic cycle; when translated into an assembly language program, each event becomes an isolated instruction sequence.

In Figure 3-1 for example, the print cycle began with a cascade of flip-flops switching on and ended with the same flip-flops switching off. In many cases a flip-flop switching on triggered one event, while the same flip-flop switching off triggered an entirely different event. Within an assembly language program, the two events will have nothing in common. Each event will be represented by a completely independent instruction sequence occurring at substantially different parts of the program.

The other major difference between digital logic and assembly language is the concept of timing. Within synchronous digital logic, as illustrated in Figure 3-1, timing is bound to clock signals and the need for clean signal interactions. Within an assembly language program, timing results strictly from the sequence in which instructions are executed. Moreover, whereas components in a digital logic circuit may switch and operate in parallel; within an assembly language program everything must occur serially.

Now the key concept to grasp from this chapter is that there is nothing innately correct about digital logic as a means of implementing anything. The fact that we have been unable to exactly duplicate digital logic using assembly language instructions does not mean that assembly language is in any way inferior; it simply means that assembly language is going to do the job in a different way.

Having spent our time in Chapter 3 drawing direct parallels between assembly language and digital logic, we will now abandon any attempt to favor digital logic. Moving on to Chapter 4, the logic illustrated in Figure 3-1 will be resimulated — but from the programmer's point of view.

:ASSIGN LOCATIONS TO DELAY COUNT TABLE

,AND TIME DURATION SELECT LINES

DELY EQU xxxx

H1H6 EQU yyyy

:TEST FOR VALID END OF PRINT CYCLE

LOP1 IN 2 :INPUT I/O PORT 2 CONTENTS TO ACCUMULATOR

RLC :SHIFT BIT 7 INTO CARRY

JNC LOP1 :IF ZERO IN CARRY, STAY IN PRINT CYCLE

:IN BETWEEN PRINT CYCLES PROGRAM EXECUTION

,INITIALLY SET I/O PORT BITS 6, 4, 3, 2 AND 1 TO 0 5 AND 0 TO 1

IN 1 :INPUT I/O PORT 1 TO ACCUMULATOR

ORI 21H :SET BITS 5 AND 0 TO 1

ANI A1H :RESET BITS 6, 4, 3, 2 AND 1 TO 0

OUT 1 :RETURN RESULTS

,TEST FOR RETURN STROBE LOW

L10 IN 2 :INPUT I/O PORT 2 TO ACCUMULATOR

ANI 10H :ISOLATE RETURN STROBE

JZ FFB :IF IT IS 0, JUMP TO FFB SIMULATION

:SIMULATION OF FFA AND ASSOCIATED LOGIC

:LOAD I/O PORT 2 CONTENTS INTO ACCUMULATOR AND

:ISOLATE BITS 1, 5 AND 6 FOR CH RDY, PW STROBE

:AND RESET, RESPECTIVELY

IN 2 :INPUT I/O PORT 2 TO ACCUMULATOR

ANI 62H :ISOLATE BITS 6, 5 AND 1

CPI 22H :IF RESET=0, CH RDY=1 AND PW STROBE=1

JNZ L10 :START NEW PRINT CYCLE, OTHERWISE RETURN TO L10

IN 1 :TO START NEW PRINT CYCLE, SET

ANI FEH :I/O PORT 1, BIT 0 TO 0

OUT 1

:NEW PRINT CYCLE SEQUENCE STARTS HERE

,SIMULATE FLIP FLOP FFB SWITCHING ON

FFB IN 1 :LOAD I/O PORT 1 INTO ACCUMULATOR

ANI FDH :RESET BIT 1 TO 0

OUT 1 :RESTORE RESULT

:SIMULATE 7411 AND GATE SWITCHING CH RDY LOW

,ALSO SIMULATE 7432 OR GATE SWITCHING PW REL LOW

IN 2 :INPUT I/O PORT 2 TO ACCUMULATOR

ANI FCH :RESET BITS 0 AND 1 TO 0

OUT 2 :RESTORE RESULT

,CH RDY LOW TURNS FFA OFF, SET BIT 0 OF I/O PORT 1 TO 1

,ALSO SIMULATE FFC AND FFD TURNING ON, SET BITS 3 AND 2 OF I/O PORT 1 TO 1

IN 1 :LOAD I/O PORT 1 INTO ACCUMULATOR

ORI 0DH :SET BITS 3, 2 AND 0 TO 1

OUT 1 :RESTORE RESULT

:PULSE START RIB MOTION HIGH

IN 2 :INPUT I/O PORT 2 TO ACCUMULATOR

ORI 8 :SET BIT 3 HIGH

OUT 2 :OUTPUT RESULT

ANI F7H :TURN BIT 3 OFF

OUT 2 :OUTPUT RESULT

:TEST VELOCITY DECODE INPUT TO CREATE PRINTWHEEL MOVE DELAY

VLDC IN 0 :INPUT I/O PORT 0 TO ACCUMULATOR

RLC :SHIFT BIT 7 INTO CARRY

JNC VLDC :STAY IN LOOP IF CARRY IS 0

Figure 3-3. The Complete Simulation Program

```

;AT END OF DELAY SIMULATE FFE SWITCHING ON
      IN      1      ;INPUT I/O PORT 1
      ANI     DFH     ;RESET BIT 5
      ORI     10H     ;SET BIT 4
      OUT     1      ;OUTPUT THE RESULT
;SIMULATE 2 MS PW SETTLING TIME DELAY
      MVI     A,0      ;LOAD ACCUMULATOR WITH 0
PWS   DCR     A        ;DECREMENT A
      JNZ     PWS      ;IF A DOES NOT DECREMENT TO 0,
                        ;RE-DECREMENT
;SIMULATE FLIP-FLOP FFF SWITCHING ON
FFF   IN      0      ;INPUT I/O PORT 0 CONTENTS TO ACCUMULATOR
      CMA      ;COMPLEMENT TO TEST FOR ALL 1 BITS
      ANI     1FH     ;ISOLATE BITS 0 THROUGH 4
      JNZ     FFF     ;IF THERE WERE ANY 0 BITS, STAY IN LOOP
      IN      1      ;INPUT I/O PORT 1 TO ACCUMULATOR
      ORI     40H     ;SET BIT 6 TO 1
      OUT     1      ;OUTPUT THE RESULT
;TEST HAMMER ENABLE FF
      IN      0      ;INPUT I/O PORT 0 TO ACCUMULATOR
      ANI     40H     ;ISOLATE BIT 6
      JZ      HPO      ;IF ZERO, BYPASS SETTING HAMMER PULSE LOW
;HAMMER ENABLE FF IS HIGH, SO HAMMER PULSE
;MUST BE OUTPUT LOW. THEREFORE SET BIT 2 OF
;/O PORT 3 TO 0
      IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
      ANI     FBH     ;SET BIT 2 TO 0
      OUT     2      ;OUTPUT RESULT
;COMPUTE TIME DELAY
HPO   LXI     H,DELY   ;LOAD DATA ADDRESS BASE INTO HL
      LDA     H1H6     ;LOAD SELECTOR INTO ACCUMULATOR
HP1   RRC      ;ROTATE ACCUMULATOR, SET CARRY TO A0
      INX     H        ;INCREMENT HL CONTENTS BY 2
      INX     H
      JNC     HP1      ;IF RRC DID NOT SHIFT 1 INTO CARRY, RETURN
      MOV     D,M      ;LOAD 16-BIT TIME DELAY CONSTANT INTO DE
      INX     H
      MOV     E,M
TDLY  DCX     D        ;EXECUTE TIME DELAY LOOP
      MOV     A,D
      ORA     E
      JNZ     TDLY
;OUTPUT HAMMER PULSE HIGH AGAIN
      IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
      ORI     4        ;SET BIT 2 TO 1
      OUT     2      ;OUTPUT RESULT
;SWITCH FLIP-FLOP FFC OFF
      IN      1      ;INPUT I/O PORT 1 TO ACCUMULATOR
      ANI     FBH     ;SET BIT 2 TO 0
      OUT     1      ;OUTPUT RESULTS

```

Figure 3-3. The Complete Simulation Program
(Continued)

```

.EXECUTE A 3 MILLISECOND TIME DELAY
    LXI    D,F7H    ;LOAD TIME CONSTANT INTO D,E
PWR1     DCX    D    ;DECREMENT D,E
        MOV    A,D    ;TEST FOR ZERO RESULT
        ORA    E
        JNZ    PWR1    ;REDECREMENT IF NOT ZERO
;SWITCH FLIP-FLOP FFD OFF
    IN     1    ;INPUT I/O PORT 1 TO ACCUMULATOR
    ANI    F7H    ;SET BIT 3 TO 0
    OUT    1    ;OUTPUT RESULTS
;EXECUTE A 2 MILLISECOND TIME DELAY
    MVI    A,83H    ;LOAD TIME CONSTANT INTO ACCUMULATOR
PWR2     DCR    A    ;DECREMENT ACCUMULATOR
        JNZ    PWR2    ;REDECREMENT IF NOT ZERO
;SET PW REL HIGH
    IN     2    ;INPUT I/O PORT 2 TO ACCUMULATOR
    ORI    1    ;SET BIT 0 TO 1
    OUT    2    ;OUTPUT RESULT
;TURN OFF FLIP-FLOPS FFB, FFE AND FFF
    IN     1    ;INPUT I/O PORT 1 TO ACCUMULATOR
    ANI    ADH    ;RESET BITS 1, 4 AND 6 TO 0
    ORI    20H    ;SET BIT 6 TO 1
    OUT    1    ;OUTPUT RESULTS
;SET CH RDY HIGH
    IN     2    ;INPUT I/O PORT 2 TO ACCUMULATOR
    ORI    2    ;SET BIT 1 TO 1
    OUT    2    ;OUTPUT RESULT
;BRANCH TO TEST FOR VALID END OF PRINT CYCLE
    JMP    LOP1
;DELAY COUNT TABLE
    ORG    DELY + 2
    pppp    ;H1 TIME DELAY
    qqqq    ;H2 TIME DELAY
    rrrr    ;H3 TIME DELAY
    ssss    ;H4 TIME DELAY
    tttt    ;H5 TIME DELAY
    uuuu    ;H6 TIME DELAY

```

The letters x, y, p, q, r, s t and u represent hexadecimal values.

Figure 3-3. The Complete Simulation Program
(Continued)

Chapter 4

A SIMPLE PROGRAM

The problems associated with simulating digital logic, as we did in Chapter 3, can be attributed to one fact: we tried to divide logic into a number of isolated transfer functions, each of which corresponded to a digital logic device. We are now going to abandon digital and combinatorial logic, pretend it does not exist and take another look at Figures 3-1 and 3-2.

ASSEMBLY LANGUAGE TIMING VERSUS DIGITAL LOGIC TIMING

Returning to Figure 3-1, simply ignore everything that exists between the left and right hand margins of the figure. What remains is a set of input signals and a set of output signals. The output signals are related to the input signals by a set of transfer functions which have nothing to do with digital logic devices.

TRANSFER
FUNCTION

The transfer functions for Figure 3-1 are loosely represented by the timing diagram in Figure 3-2. What does "loosely represented" mean? It means that timing which relates to system requirements is mixed indiscriminately with timing that simply reflects the needs of digital logic. We can abandon timing considerations that simply reflect the needs of digital logic. To be specific, the printhead must still be fired by outputting one of six solenoid pulses; the various movement and settling delays must also be maintained. But we can abandon time delays that separate one signal's change of state from another simply to keep the digital logic clean.

From the programmer's point of view, therefore, the timing diagram illustrated in Figure 4-1 is a perfectly valid substitution for the logic designer's timing diagram illustrated in Figure 3-2.

INPUT AND OUTPUT SIGNALS

Looking at Figure 4-1 you will see that we have abandoned a lot more than minor timing delays; we have also abandoned most of our signals. But there is a simple criterion for determining whether a signal is really necessary within a microcomputer system. This is the criterion: if the signal is uniquely associated with real time events in logic external to the microcomputer system, then the signal must remain. If the source and destination of the signal are within the microcomputer system "black box", then the signal may be abandoned. Based on this criterion, let us take another look at our input and output signals.

First consider the input signals.

RETURN STROBE and **PW STROBE** are meaningless signals. As digital logic, these two signals are print cycle sequence initiators. Within an assembly language program, jumping to the first instruction of a sequence is all the initiation you need. The fact that **RETURN STROBE** represents a print cycle during which the printhead is not fired is unimportant, because **HAMMER ENABLE** is used to actually suppress **HAMMER PULSE**.

INPUT
SIGNALS

We will combine the various hammer firing inhibit signals into one hammer status input. There are five such signals: PFL REL, RIB LIFT RDY, RIBBON ADVANCE, PFR REL and CA REL. Each of these signals owes its origin to different logic external to Figure 3-1; in the digital logic implementation, these signals are ANDed in order to create a master HAMMER INTERLOCK signal. In our assembly language implementation we will wire OR all of these external signals to a single pin which becomes a HAMMER INTERLOCK status.

RESET will be maintained as a master RESET signal tied to the CPU RESET pin. RESET can therefore be ignored by the assembly language program, however, recall that once RESET is activated, program execution is going to resume with the instruction stored at memory location 0.

EOR DET will be maintained. This is the signal which detects end of ribbon and prevents a print cycle from ever ending, thus inhibiting further character printing after the ribbon is exhausted.

HAMMER ENABLE FF must be maintained; it suppresses the printhead firing pulse during printwheel repositioning print cycles.

The function performed by the six hammer pulse length signals, H1 through H6, must remain, but the signals themselves will disappear. Instead of using six pins of an I/O port to identify hammer pulse width, we are going to create time delays directly from ASCII character codes

Let us now turn our attention to the output signals.

To begin with, we can eliminate all of the flip-flop outputs. The boundary of each time interval within the print cycle is already identified by an existing signal changing state. If more than one external logic event must be triggered by a transition from one time interval to the next, there is nothing to stop the appropriate signal from being buffered externally, then used to trigger numerous external logic events. Within the microcomputer program, there is no reason why duplicate signals should be output simply to identify the transition from one print cycle time interval to the next.

The remaining output signals are maintained. It is possible that some of these signals would disappear if additional external logic were replaced by more assembly language programs within the microcomputer system, but given the bounds of the problem, as stated, the remaining signals are needed in order to define the print cycle time intervals.

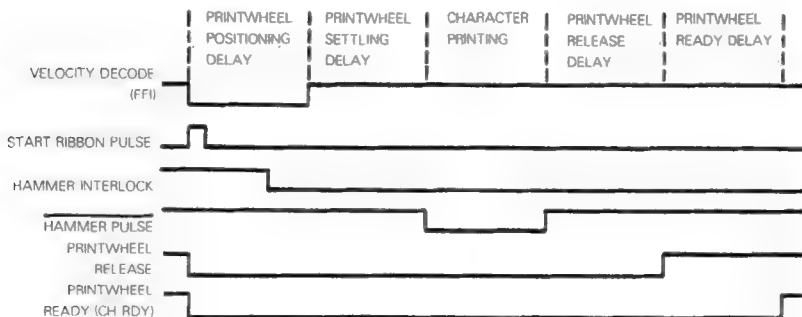
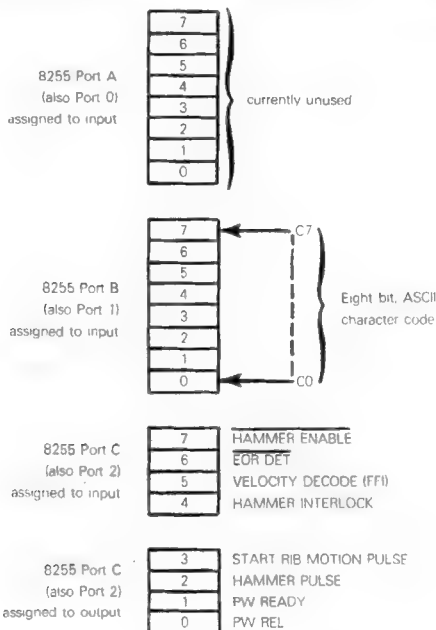


Figure 4-1. Timing For Figure 3-1, From The Programmer's Viewpoint

Given our new, simplified set of signals, we can eliminate the 8212 I/O buffer and use one 8255 Programmable Peripheral Interface (PPI), operating in Mode 0, with I/O ports and pins assigned as follows:

PIN ASSIGNMENTS



MICROCOMPUTER DEVICE CONFIGURATION

We are now in a position to select the devices needed for program implementation. The selection is really quite straightforward; in addition to the CPU, we will need one 8255 Programmable Peripheral Interface, some read-only memory for program storage and some read-write memory for general data storage. The CPU, in reality, consists of three devices, the CPU itself, the 8224 Clock chip and the 8228 Bus Controller. Combining these devices, Figure 4-2 illustrates the microcomputer system which results. Now if you don't immediately understand Figure 4-2 do not despair, there are only a few aspects of this figure which are consequential to our immediate discussion.

GENERAL DESIGN CONCEPTS

This is the most important concept to derive from Figure 4-2: when designing logic by writing assembly language programs within a microcomputer system, the program you write is going to be highly dependent upon the device configuration. There is nothing unique about the way in which devices have been combined as illustrated in Figure 4-2, alternative configurations would be equally viable. The assembly language programs created, however, might differ markedly from one microcomputer configuration to the next and this is a factor you should not lose sight of when writing microcomputer programs. Also, do not be afraid of modifying the selected hardware configuration; that is precisely what we will do in

Chapter 5. **Microcomputer device configuration and assembly language programming interact strongly and should not be separated.** These two steps should be within one iterative loop. During the early stages of writing a microcomputer program, you should assume that in the course of writing the assembly language program, you will discover features of the hardware that can be improved, that in turn means the program will have to be rewritten.

This is a good point to bring up one of the reasons why higher level languages are not desirable when you are programming a microcomputer to replace digital logic. Higher level languages are

**HIGHER
LEVEL
LANGUAGES**

problem-oriented. For example, it is hard to look at a PL/M program statement and visualize the exact way in which data will be moved around a microcomputer system in response to the statement's execution. It is even harder to relate PL/M programs to exact device configurations. Assembly language, on the other hand, has a one-for-one relationship with your hardware.

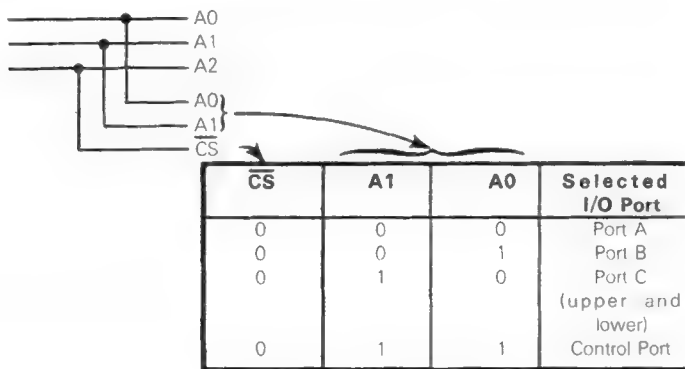
8255 PROGRAMMABLE PERIPHERAL INTERFACE (PPI)

Now let us turn our attention to the specific way in which devices have been incorporated into Figure 4-2.

The 8255 Programmable Peripheral Interface will respond to I/O port addresses 0, 1 and 2 for I/O Ports A, B and C, respectively. This is because the chip select is tied to address bus line A2

**I/O PORT
SELECT**

Since the I/O port address is output on the eight low order lines of the address bus when an IN or OUT instruction is executed, the 8255 PPI in Figure 4-2 will respond to I/O port addresses as follows:



CS must be 0 for the PPI to be selected. This means that as shown in Figure 4-2, the PPI will be selected whenever A2 is 0, regardless of what A3 through A7 may be. Thus **the PPI will respond to any I/O port address, excluding 4, 5, 6 and 7:**



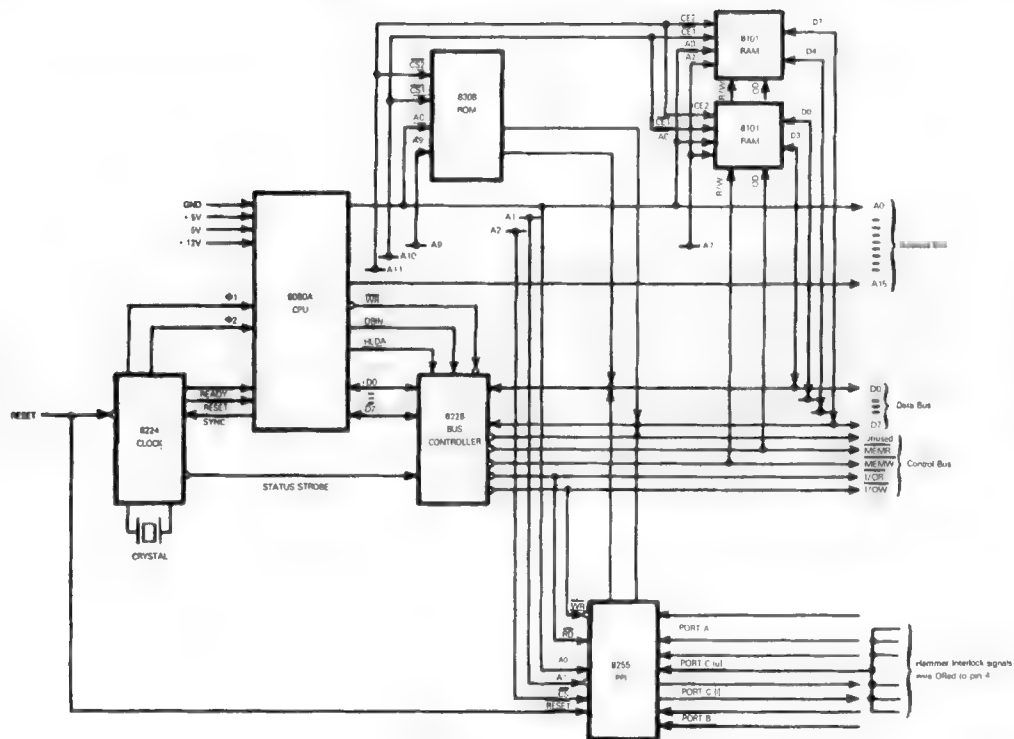


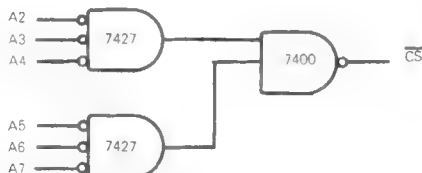
Figure 4-2 Microcomputer Configuration

If a microcomputer configuration contains a large number of Programmable Peripheral Interfaces (PPIs), the chip select logic may become a little more complex. If a PPI is to respond to four unique I/O port addresses, excluding all others, then the chip select input must be created by combining the six high order address lines in some unique way. This, of course, implies that the microcomputer system is going to contain 64 PPIs,— and that is unlikely.

CHIP SELECT IN SIMPLE SYSTEMS

Suppose the 8255 PPI in Figure 4-2 must respond to I/O port addresses 0, 1, 2 and 3, only. This is one way of creating the Chip Select (\overline{CS}) input:

CHIP SELECT IN LARGER SYSTEMS



If, and only if all six address lines A2 through A7 are low, \overline{CS} will be low, and the 8255 PPI will be selected.

Now the data direction and port utilization illustrated for the 8255 PPI in Figure 4-2 is not a hardware feature. At any time port utilization may be modified by writing the appropriate control word into I/O Port 3, which is the control port for the 8255 PPI as configured.

The RESET logic also needs some comment. Instead of testing for a reset condition in between print cycles, as we did in Chapter 3, **we are going to use a hardware RESET signal**, but in a microcomputer environment.

RESET LOGIC

The RESET signal, being connected to the 8224 Clock and the 8255 PPI, will clear all registers in the 8255 PPI and the Program Counter in the 8080 CPU. This means that program execution will restart with the instruction stored in the memory byte whose address is 0. **We must therefore have post-reset and system initialization program steps beginning at this memory location.**

SYSTEM INITIALIZATION

When the system is initialized, "in between print cycles" conditions must be re-established immediately. These are the necessary steps:

- 1) If the printhead has been fired, discontinue the firing pulse and allow the printhead time to retract.
- 2) Move the printwheel back to its position of visibility.
- 3) Insure that output signals have their "in between print cycles" status.

We now arrive at another fundamental programming concept: **there is a "most efficient" sequence in which you should write assembly language source programs.** We could go ahead and write an initialization program to implement a RESET, but that would require a lot of guessing. How do we know that the printhead has been fired? How do we move the printwheel back to its position of visibility? RESET is going to abort a print cycle — therefore the print cycle program must be created before we can know how to abort it.

PROGRAM IMPLEMENTATION SEQUENCE

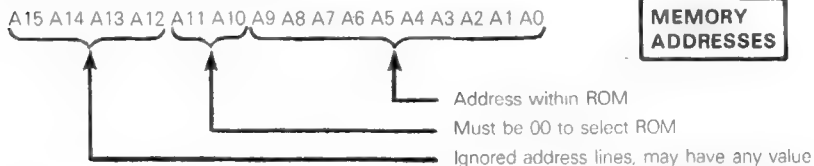
Generally stated, you should start writing a program by implementing the most important event in your logic, then you should work away from this beginning, implementing dependent events.

Specifically, we are going to postpone creating a program to implement the RESET logic until the print cycle program has been created.

ROM AND RAM MEMORY

The use of the RESET signal means that there must be a memory byte with address 0. In Figure 4-2 this memory byte will be part of the 8308 ROM.

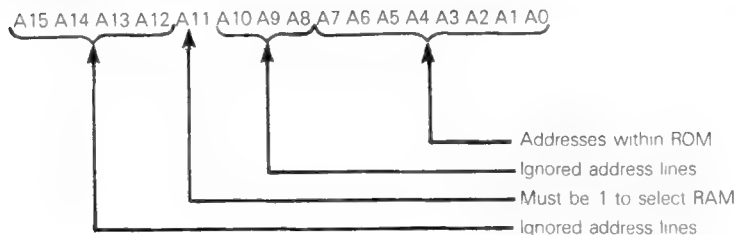
This is a 1024-byte ROM. It will respond to memory addresses 0 through 03FF₁₆, since its chip select lines are tied to address bus lines A10 and A11. This may be illustrated as follows



Once again we are using a primitive ROM chip select on account of the microcomputer system's simplicity. We define the address range 0 through 03FF₁₆ for the 0124 bytes of ROM memory; but in fact, a wide variety of other addresses would also access ROM memory — the address lines A12 through A15 can have any value. Providing A10 and A11 are both 0, ROM memory will be accessed. There is nothing to prevent you from selecting memory in this primitive way, providing yours is a small microcomputer system. There is no reason why you should incur additional expense creating complex chip select codes using all of the high order six address bus lines. Even from the programming point of view, you will not have to rewrite programs should you expand your system and include more memory at a later date. Providing you do not now use any of the alternative addresses that would also select the ROM, then at some future time you could take one of these alternative sets of addresses, use it to select another ROM, and in no way impact programs already written.

By specifying ROM for program storage, we are assuming that the product will be developed in sufficient volume to justify the expense of creating a ROM mask. If your volume does not justify the expense of creating ROM, then you can use Programmable Read Only Memory (PROM)

The two 8101 RAM devices each provide 1024-bits of read/write memory, organized into 256 4-bit units. Each RAM therefore provides half of a read/write memory byte. The 256 bytes of RAM will have addresses 0800₁₆ through 08FF₁₆. This may be illustrated as follows:



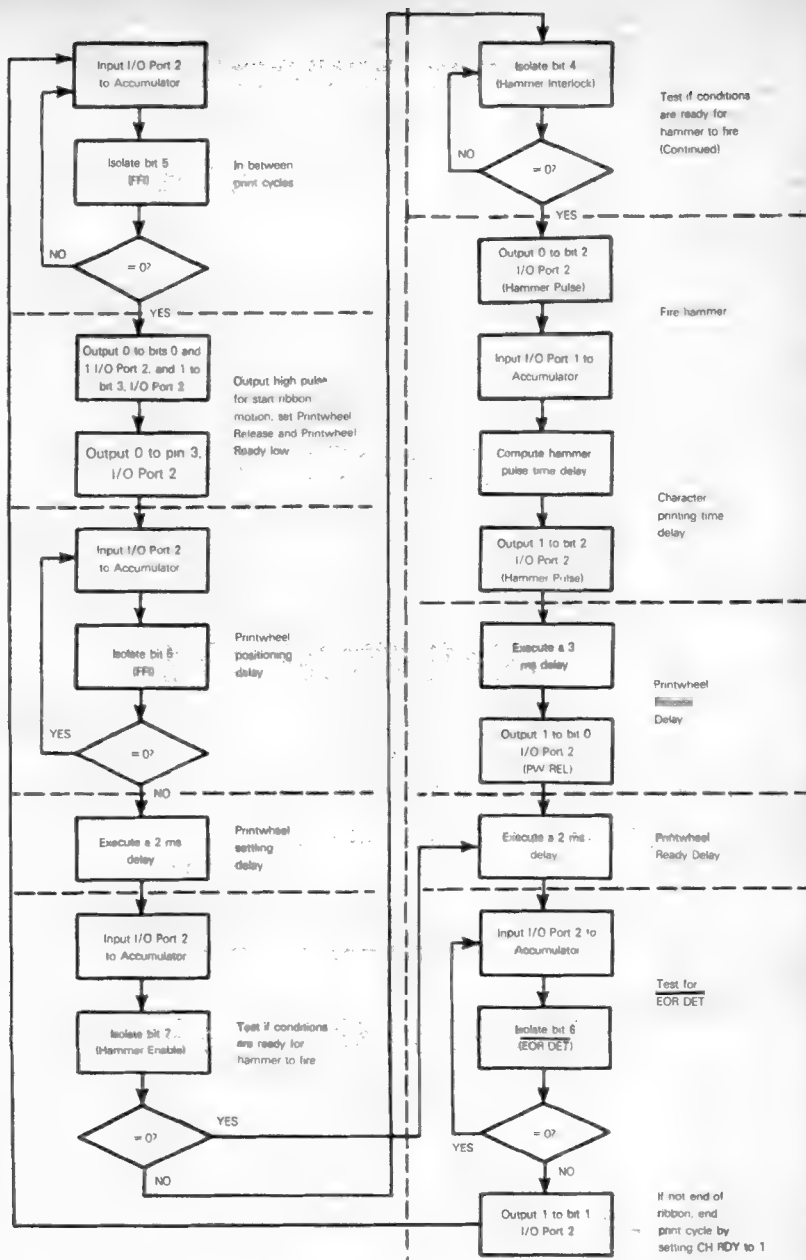


Figure 4-3. First Attempt At Program Flowchart

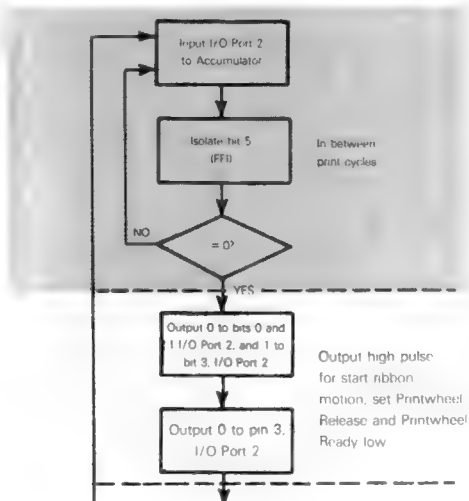
Even though memory addresses 0800₁₆ through 08FF₁₆ have been specified as providing the RAM address space, once again a large number of other addresses would also select RAM. Note however, that in no case will a RAM address coincide with a ROM address; address bus line A11 must always be 0 to select ROM, while it must always be 1 to select RAM. Address contentions will therefore never arise.

Other features of Figure 4-2 are not significant to program generation at the level we are currently discussing, therefore you do not need to understand the hardware configuration in any further detail.

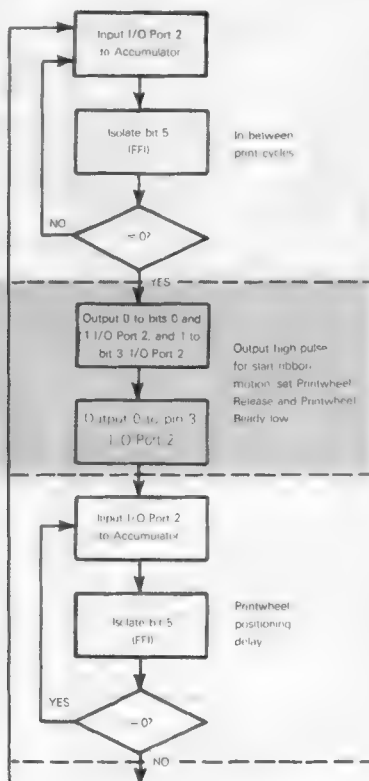
PROGRAM FLOWCHART

Let us now turn our attention to the functions which must be performed by the microcomputer system. These functions are identified by the flow chart illustrated in Figure 4-3. We will analyze this flow chart, step-by-step.

We are going to use the velocity decode input signal (FFI) to identify the start of a new print cycle. In between print cycles, therefore, the program continuously inputs I/O Port 2 contents to the Accumulator, testing bit 5. So long as this bit equals 1, a new print cycle has not begun. As soon as this bit equals 0, a new print cycle is identified:

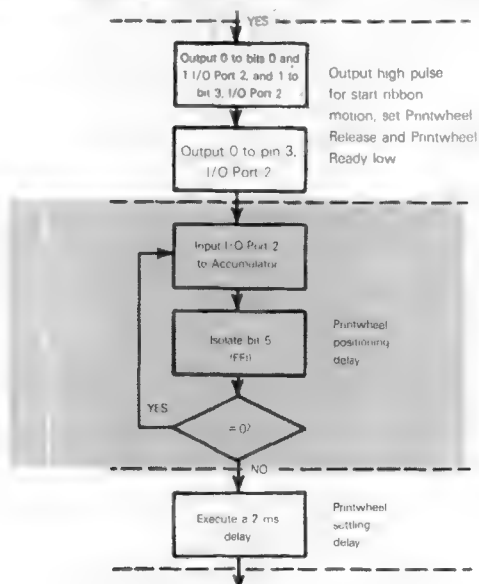


The first thing that happens within the new print cycle is that a **high START RIBBON MOTION pulse is output by sequentially writing a 1, then a 0 to bit 3 of I/O Port 2. Also, 0s are output to bits 0 and 1 of I/O Port 2**, since PRINTWHEEL RELEASE and PRINTWHEEL READY must both be output low at the start of the print cycle:

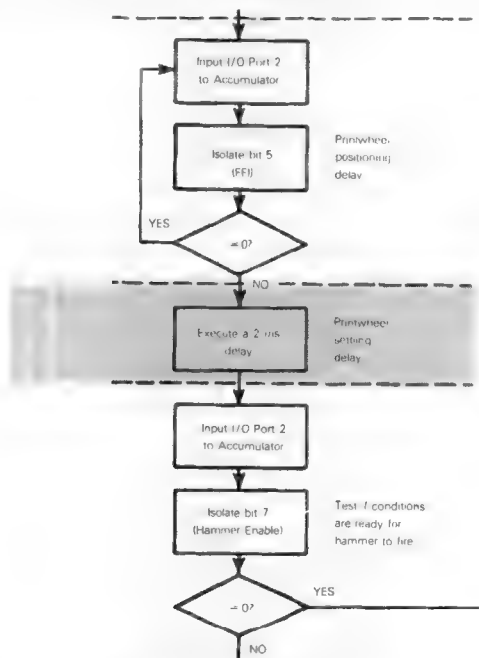


The printwheel positioning delay is computed by the velocity decode signal FFI. So long as this signal is low, the printwheel is still being positioned. We therefore go into a variable delay loop, which in terms of program logic is the inverse of the "in between print cycles" delay loop. Once again, I/O Port 2 contents are input to the Accumulator and bit 5 is tested; however,

we stay in the delay loop until bit 5 is 1. At that time the printwheel positioning delay is over. :

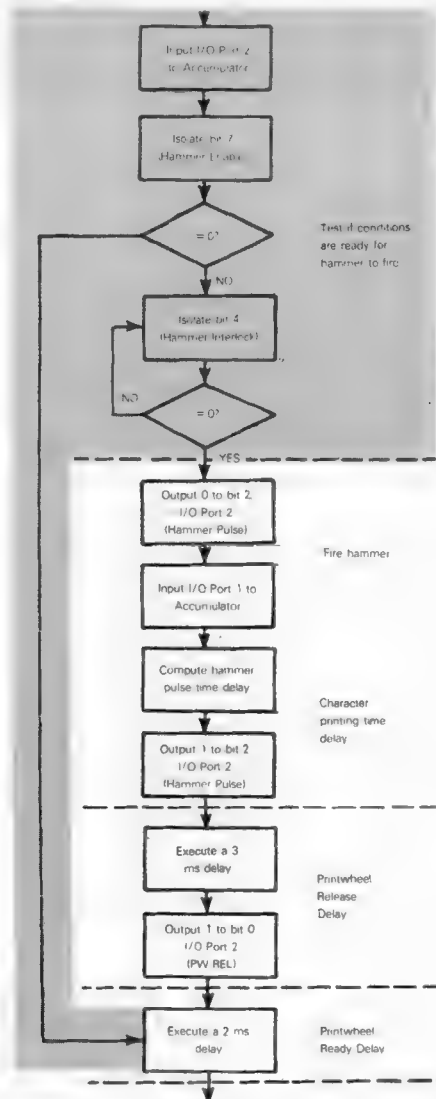


The printwheel positioning delay must be followed by a 2 millisecond printwheel settling delay. The usual delay loop will be executed here:

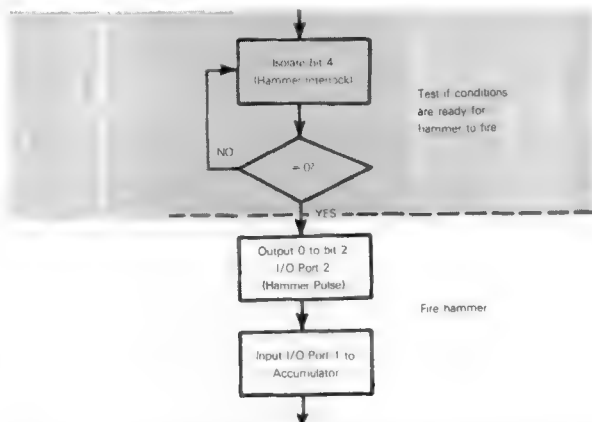


At the end of the printwheel settling delay, the printhammer is fired, providing the **HAMMER INTERLOCK** signal is low and **HAMMER ENABLE** is high. Recall that HAMMER INTERLOCK is a single status bit, used by all external conditions that can prevent the hammer from being fired. Any signal inputting a high level to this status pin will suppress printhammer firing.

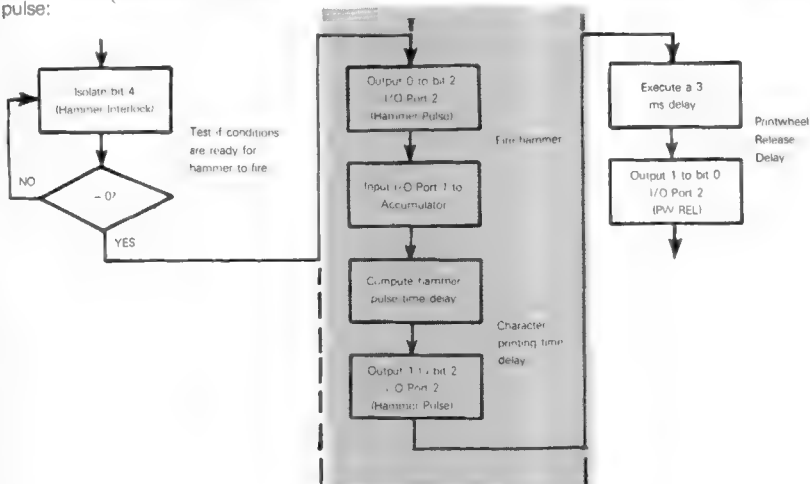
A printwheel repositioning print cycle is identified by HAMMER ENABLE being input low. This condition is detected by isolating bit 7 of I/O Port 2 before testing the condition of HAMMER INTERLOCK. If bit 7 of I/O Port 2 equals 0, then the entire printhammer firing sequence is skipped and we jump directly to the printwheel ready delay, which is the last time delay of the print cycle.



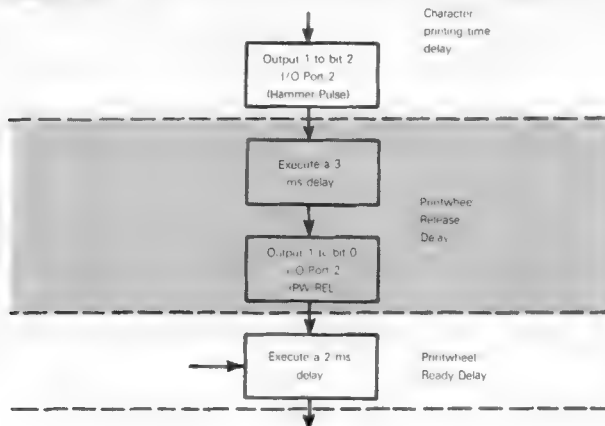
If **HAMMER ENABLE** is high, this is a character printing cycle, so the printhammer will be fired, but only when HAMMER INTERLOCK is 0. So long as any of the signals wire-ORed to pin 4 of I/O Port 2 is high, the program will stay in an endless loop, continuously testing the status of this I/O port pin. When finally the I/O port pin equals 0, the program will advance to the printhammer firing instruction sequence:



In order to fire the printhammer, a variable length firing pulse must be output. To do this a 0 is output to pin 2 of I/O Port 2, since this is the pin via which the hammer pulse is output. Next the hammer pulse time delay is computed. We will describe how the hammer pulse width is computed after completing a description of the flow chart. At the end of the printhammer firing time delay, a 1 is output to bit 2 of I/O Port 2. This terminates the printhammer firing pulse:

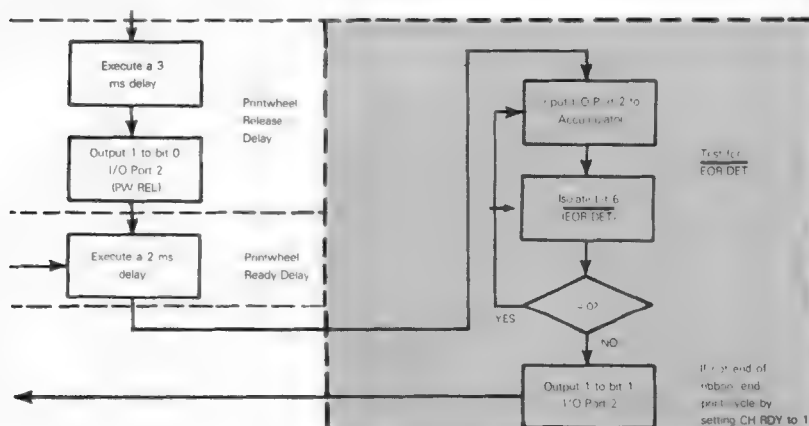


Now two settling delays follow. First there is a 3 millisecond printwheel release delay, the termination of which is marked by a 1 being output to bit 0 of I/O Port 2. This causes PW REL to output high:



Next, a 2 millisecond printwheel ready delay is executed. The end of this delay and the end of the print cycle is marked by a 1 output to bit 1 of I/O Port 2; this sets CH RDY high. **We do not want to do this, however, if there is an end-of-ribbon status.** This status is identified by EOR DET being low

The program therefore inputs I/O Port 2 and isolates bit 6, via which EOR DET is input to the microcomputer system. If EOR DET equals 0, then the program stays in an endless loop, continuously retesting bit 6 of I/O Port 2, thus another print cycle cannot begin. Only if EOR DET is detected equal to 1 will the print cycle terminate with CH RDY set to 1:

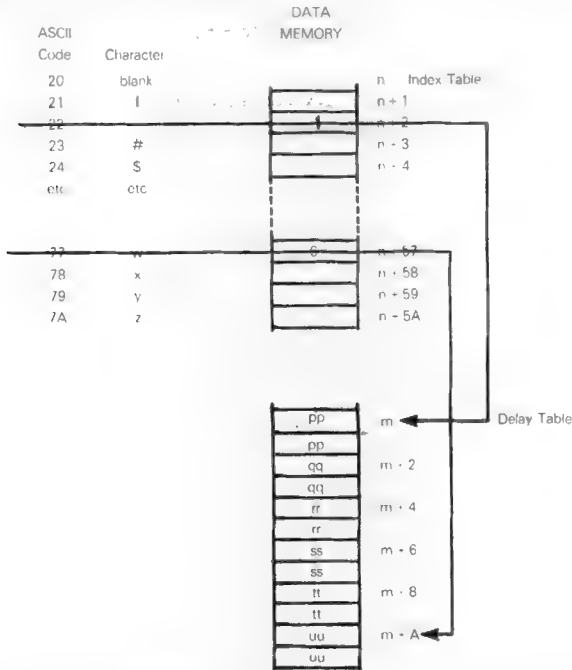


Now let us turn our attention to the method via which the appropriate printhammer firing delay is computed. In Figure 3-1, the appropriate printhammer firing delay was signaled by one of six lines (H1 through H6) being input true. Some external logic had to generate the true line, based on the nature of the character being printed, **this kind of operation is easier to do within a microcomputer program.**

PRINTHAMMER FIRING DELAY

This is the method we will use to compute the appropriate printhammer firing pulse time delay: every character to be printed is represented by one ASCII code data byte, as illustrated in Appendix A.

If we ignore the high order parity bit, then 128 possible bit combinations remain. If you look at the ASCII codes given in Appendix A, you will see that only character codes between 20_{16} and $7A_{16}$ are significant. Therefore, only $5A_{16}$ (or 90_{10}) code combinations need to be accounted for. Each of these code combinations will have assigned to it one byte in a 90-byte table, and in this byte will be stored a number between 1 and 6. This number will identify the time delay required by the character. A 12 byte table will contain the six actual time delays associated with the six digits. This scheme may be illustrated as follows:



In the above illustration the letters "n" and "m", to the right of the data memory, represent any valid base memory addresses. For example, "n" might represent 0380_{16} while "m" represents $03F0_{16}$.

Consider two examples.

ASCII code 22_{16} signifies the double quotes character ("), which requires the shortest time delay. The data memory byte with address $n + 2$ corresponds to this ASCII code. 1 is stored in this data memory byte. Therefore, the first time delay, represented by pppp, is the value which must be loaded into the D and E registers before executing the long time delay loop which creates the printhammer firing pulse for the " character.

ASCII code 77_{16} represents "w". The data memory byte with address $n + 57_{16}$ corresponds to this ASCII code. Within this data memory byte the value 6 is stored, which means that the longest printhammer firing delay is required for a "w". Therefore, a value represented by uuuu will be loaded into the D and E registers before executing the long time delay loop which creates the printhammer firing pulse for the w character.

Figure 4-4 identifies the program steps via which the printhammer firing delay will be computed.

In order to better understand Figure 4-4, we will go down steps ① through ⑩ for the case of "w".

- ① The ASCII representation of lower case w is input to the Accumulator

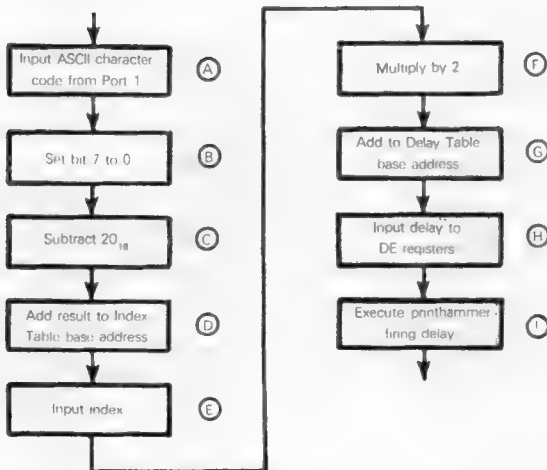


Figure 4-4. Program Flow Chart To Compute Printhammer Firing Pulse Length

- (B) We must set the parity bit to 0. To do this the Accumulator contents are ANDed with 7FH



- (C) The index table entry corresponding to lower case w is computed by adding the ASCII code, less 20_{16} to the index table base address. We must subtract 20_{16} because the first 16 codes have no ASCII equivalent:



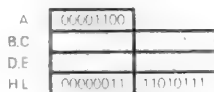
- (D) The index table base address is loaded into the H and L registers. We will assume this address is 0380_{16} . Then the Accumulator contents are added to this 16-bit address



- (E) The appropriate index is loaded from the index table into the Accumulator



- (F) Since the actual delay is two bytes long, we are going to calculate the address of the appropriate delay by adding twice the index to the delay table base address. First we multiply the index by 2



- (G) Next we add the index multiplied by 2, to the delay table base address. Assume this base address is $03F0_{16}$. This base address is again loaded into the H and L registers, after which the Accumulator contents are added to the H and L registers' contents:



- (H) The two bytes addressed by H and L are loaded into the D and E registers



- (I) The D and E registers now contain the correct initial value for a long delay to be executed as described in Chapter 2


```

.PRINT CYCLE PROGRAM
;IN BETWEEN PRINT CYCLES TEST FF1 (BIT 5 OF I/O
;PORT 2 FOR A 0 VALUE)
START    IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
        ANI      20H    :ISOLATE BIT 5
        JNZ      START  :IF NOT 0, RETURN TO START
;INITIALIZE PRINT CYCLE. OUTPUT 0 TO BITS 0
;AND 1 OF I/O PORT 2. OUTPUT 1 TO BIT 3 OF
;I/O PORT 2
        MVI      A,0CH  :LOAD MASK INTO ACCUMULATOR
        OUT      2      :OUTPUT TO I/O PORT 2
;OUTPUT 0 TO BIT 3 OF I/O PORT 2. THIS COMPLETES
;START RIBBON MOTION PULSE
        MVI      A,4    :LOAD MASK INTO ACCUMULATOR
        OUT      2      :OUTPUT TO I/O PORT 2
;TEST FOR END OF PRINTWHEEL POSITIONING
LOP1:    IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
        ANI      20H    :ISOLATE BIT 5
        JZ       LOP1   :IF 0 RETURN TO LOP1
;EXECUTE PRINTWHEEL SETTLING 2 MS DELAY
        MVI      A,0    :LOAD ACCUMULATOR WITH 0
LOP2:    DCR      A      :DECREMENT A
        JNZ      LOP2   :IF A DOES NOT DECREMENT TO 0, RE-DECREMENT
;TEST PRINTHAMMER FIRING CONDITIONS
LOP3:    IN      2      :INPUT I/O PORT 2 TO ACCUMULATOR
        RLC       :MOVE BIT 7 INTO CARRY
        JNC      PRD    :IF CARRY IS 0, BYPASS PRINTHAMMER FIRING
        ANI      20H    :ISOLATE BIT 4 WHICH IS NOW BIT 5
        JZ       LOP3   :WAIT FOR NONZERO VALUE BEFORE FIRING
;FIRE PRINTHAMMER
        IN      2      :SET HAMMER PULSE LOW OUTPUT 0
        ANI      FBH    :TO BIT 2 OF I/O PORT2
        OUT      2
        IN      1      :INPUT ASCII CHARACTER TO ACCUMULATOR
        ANI      7FH    :MASK OUT HIGH ORDER BIT
        SUI      20H    :SUBTRACT 20H
        LXI      H,INDX :LOAD INDEX TABLE BASE ADDRESS TO HL
        ADD      L      :ADD ACCUMULATOR CONTENTS TO HL
        MOV      L,A     :LOAD INDEX INTO ACCUMULATOR
        ADD      A      :MULTIPLY BY 2
        LXI      H,DELAY :LOAD DELAY TABLE BASE ADDRESS INTO HL
        ADD      L      :ADD ACCUMULATOR CONTENTS TO HL
        MOV      L,A     :LOAD DELAY CONSTANT INTO D.E
        INX      H
        MOV      D,M     :LOAD DELAY CONSTANT INTO D.E
LOP4:    DCX      D      :EXECUTE LONG DELAY
        MOV      A,D
        ORA      E
        JNZ      LOP4

```

Figure 4-5. A Simple Print Cycle Instruction Sequence
Without Initialization Or Reset

```

IN      2      ;AT END OF DELAY OUTPUT 1 TO BIT 2
ORI     4      ;OF I/O PORT 2 (HAMMER PULSE HIGH)
OUT     2

;EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
LXI     D,F7H  ;LOAD TIME CONSTANT INTO D,E
LOP5    DCR     D      ;DECREMENT D,E
MOV     A,D     ;TEST FOR 0 IN D,E
ORA     E
JNZ     LOP5    ;REDECREMENT IF NOT 0
;OUTPUT 1 TO BIT 0 OF I/O PORT 2. THIS SETS
;PW REL HIGH
IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
ORI     1      ;SET BIT 0 TO 1
OUT     2      ;OUTPUT RESULT

;EXECUTE A 2 MILLISECOND PRINTWHEEL READY DELAY
PRD     MVI     A,0    ;LOAD ACCUMULATOR WITH 0
LOP6    DCR     A      ;DECREMENT A
JNZ     LOP6    ;IF A DOES NOT DECREMENT TO 0, RE-DECREMENT

;TEST FOR EOR DET (BIT 6 OF I/O PORT 2) EQUAL
;TO 0 AS A PREREQUISITE FOR ENDING THE PRINT CYCLE
LOP7    IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
ANI     40H     ;ISOLATE BIT 6
JZ      LOP7    ;RETURN AND RETEST IF 0

;AT END OF PRINT CYCLE SET BIT 1 OF I/O PORT 2 TO 1
;THIS SETS CH RDY HIGH
IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
ORI     2      ;SET BIT 1 TO 1
OUT     2      ;OUTPUT RESULT
JMP     START   ;JUMP TO NEW PRINT CYCLE TEST

```

Figure 4-5. A Simple Print Cycle Instruction Sequence
Without Initialization Or Reset (Continued)

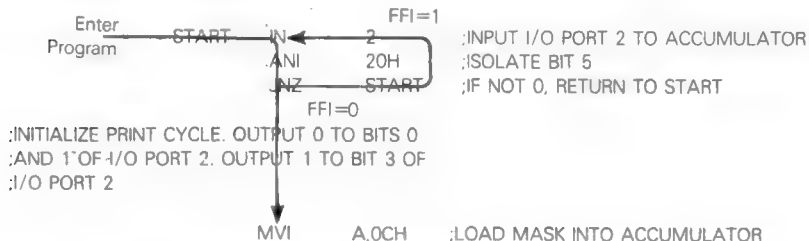
Putting together the program flow charts illustrated in Figures 4-3 and 4-4, we generate the entire required program, as illustrated in Figure 4-5. This program is now described, section-by-section.

In between print cycles the following three-instruction loop continuously tests the status of I/O Port 2, bit 5. The FFI signal is input to this pin. So long as this signal is input high, a new print cycle cannot start. As soon as this signal is input low, the printwheel is identified as being in motion — which means that a new print cycle is underway:

;PRINT CYCLE PROGRAM

;IN BETWEEN PRINT CYCLES TEST FFI (BIT 5 OF I/O

;PORT 2 FOR A 0 VALUE)



As soon as a new print cycle starts, the **PRINTWHEEL RELEASE** and **PRINT-WHEEL READY** signals must be output low. Also, a high start ribbon motion pulse must be output so that when the printhead fires, fresh ribbon is in front of the character which is to be printed. These initial signal changes may be illustrated as follows:

:INITIALIZE PRINT CYCLE. OUTPUT 0 TO BITS 0

:AND 1 OF I/O PORT 2. OUTPUT 1 TO BIT 3 OF

:I/O PORT 2

MVI A,0CH

:LOAD MASK INTO ACCUMULATOR

OUT 2

:OUTPUT TO I/O PORT 2

:OUTPUT 0 TO BIT 3 OF I/O PORT 2. THIS COMPLETES

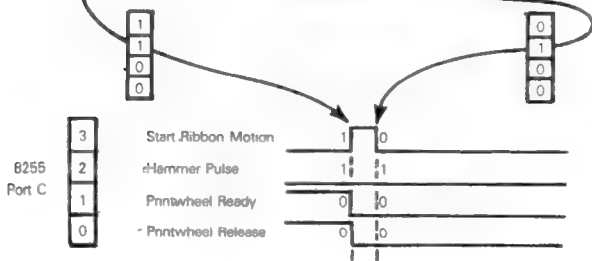
:START RIBBON MOTION PULSE

MVI A,0

:LOAD MASK INTO ACCUMULATOR

OUT 2

:OUTPUT TO I/O PORT 2



In the above illustration, notice that I/O Port C, pin 2 has been forced to output 1. This is the HAMMER PULSE pin, which goes low only for the duration of the printhead firing pulse. At this point in the print cycle, this signal is high, so outputting 1 is harmless.

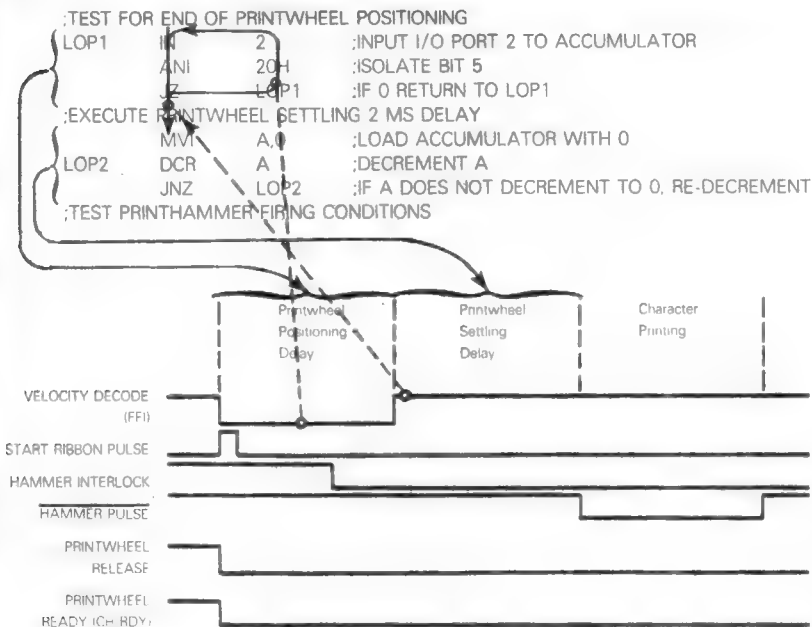
**PROGRAMMED
SIGNAL PULSE**

The program now executes a variable length delay, during which time the printwheel either moves until the appropriate character petal is in front of the printhead, or the printwheel moves back to its position of visibility. In either

**TIME DELAY
OF VARIABLE
LENGTH**

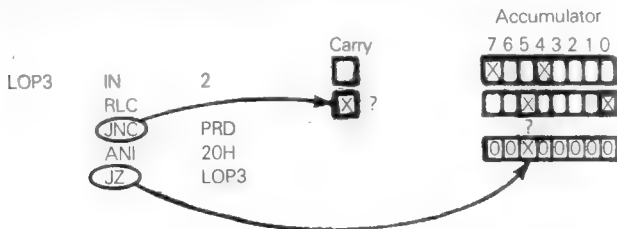
case external logic inputs signal FFI low for the duration of the printwheel positioning delay. As

soon as the printwheel has been positioned, FFI is detected high and program **logic advances to the 2 millisecond printwheel settling delay**. We have seen this three-instruction delay loop frequently before:



Now the printhammer is ready to be fired. First we test the condition of HAMMER ENABLE, which has been connected to pin 7 of I/O Port 2. If this signal is low, then we are in a printwheel repositioning print cycle and the entire hammer firing instruction sequence is bypassed. Notice that the condition of bit 7 is tested by shifting into the Carry status. **If HAMMER ENABLE is high, we pass this test. But HAMMER INTERLOCK must still be tested;** this signal is input to I/O Port 2, pin 4.

The Shift instruction moved bit 4, (representing the HAMMER INTERLOCK) to bit 5 and bit 7 (representing HAMMER ENABLE) into the Carry status:



Having loaded the contents of I/O Port 2 into the Accumulator once, we have serially tested the condition of two bits. Each bit could have been tested individually via the following six instructions:

```

                IN      2      ;INPUT I/O PORT 2 TWO ACCUMULATOR
                ANI     80H    ;ISOLATE BIT 7
                JZ      PRD    ;IF BIT 7 IS ZERO BYPASS PRINTHAMMER FIRING
LOP3           IN      2      ;INPUT I/O PORT 2 TO ACCUMULATOR
                ANI     10H    ;ISOLATE BIT 4
                JZ      LOP3   ;WAIT FOR NONZERO VALUE BEFORE FIRING

```

If HAMMER ENABLE is detected low, execution branches to the instruction labeled PRD. You will find this instruction close to the end of the program, at the beginning of the instruction sequence which executes **a 2 millisecond PRINTWHEEL READY delay.**

Note that the five-instruction sequence illustrated in Figure 4-5 tests for HAMMER ENABLE low within the loop that tests for HAMMER INTERLOCK high. Now HAMMER ENABLE will be either high or low for the duration of the print cycle; it will not change level during the print cycle. Therefore the fact that it is continuously being tested is redundant. It serves no purpose, but it does no harm.

Next the printhammer is fired. The instruction sequence which causes the printhammer to fire implements steps ④ through ⑩, which we have already described. In order to make the instruction sequence easier to understand, it is reproduced below with labels ④ through ⑩ added:

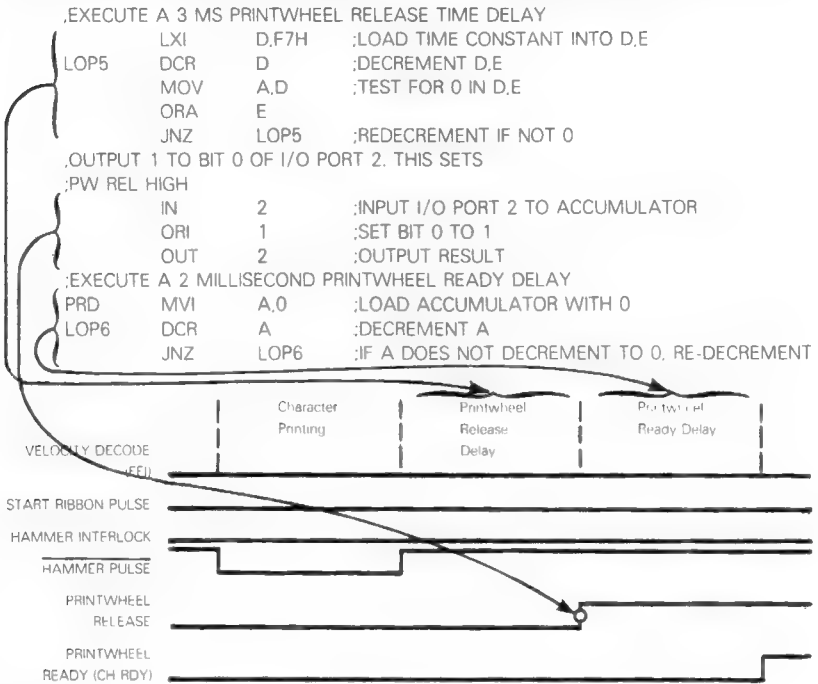
```

;FIRE PRINTHAMMER
                IN      2      ;SET HAMMER PULSE LOW. OUTPUT 0
                ANI     FBH    ;TO BIT 2 OF I/O PORT2
                OUT     2
④             IN      1      ;INPUT ASCII CHARACTER TO ACCUMULATOR
⑤             ANI     7FH    ;MASK OUT HIGH ORDER BIT
⑥             SUI     20H    ;SUBTRACT 20H
⑦             LXI     H,INDX  ;LOAD INDEX TABLE BASE ADDRESS TO HL
⑧             ADD     L      ;ADD ACCUMULATOR CONTENTS TO HL
                MOV     L,A
⑨             MOV     A,M     ;LOAD INDEX INTO ACCUMULATOR
                ADD     A      ;MULTIPLY BY 2
                LXI     H,DELY  ;LOAD DELAY TABLE BASE ADDRESS INTO HL
⑩             ADD     L      ;ADD ACCUMULATOR CONTENTS TO HL
                MOV     L,A
                MOV     E,M     ;LOAD DELAY CONSTANT INTO D,E
⑪             INX     H
                MOV     D,M
LOP4           DCX     D      ;EXECUTE LONG DELAY
                MOV     A,D
⑫             ORA     E
                JNZ     LOP4
                IN      2      ;AT END OF DELAY OUTPUT 1 TO BIT 2
                ORI     4      ;OF I/O PORT 2 (HAMMER PULSE HIGH)
                OUT     2

```

;EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY

A 3 millisecond PRINTWHEEL RELEASE time delay is now executed and the end of this time delay is marked by the PRINTWHEEL RELEASE signal being output high. Next, the 2 millisecond PRINTWHEEL READY delay is executed:



Before terminating the print cycle by outputting PRINTWHEEL READY (CH RDY) high, the program must insure that the end of ribbon has not been reached. If EOR DET is detected low the program stays in an endless loop until the ribbon has been changed, then EOR DET will be input high by external logic

When EOR DET is detected high, the final instructions of the program set PRINTWHEEL READY high, then return to the beginning of the program and wait for the next print cycle

PROGRAM LOGIC ERRORS

The program we have developed in this chapter contains a logic-error which could not occur in a digital logic implementation. The error is in the hammer pulse time delay computation.

In a digital logic implementation, the ASCII code for any character would be processed as seven individual signals. These signals would be combined in some way to generate one of the time delay signals H1 through H6. **It does not matter what ASCII code combination is input, one of the time delay signals H1 through H6 will be output high;** if the signal generation logic is unsound, a time delay signal will still be created, although it may be the wrong signal.

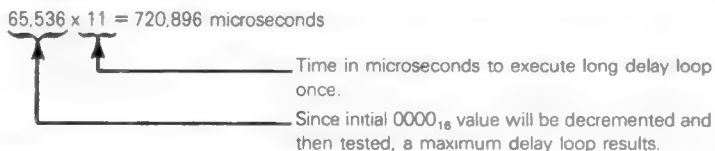
Now look at the assembly language program implementation. It is simple enough for us to look up the table in Appendix A and see that valid ASCII codes only cover the range 20₁₆

**LIMIT
CHECKING**

through 7A₁₆. That does not prevent a logic designer from using the microcomputer system we create in a special system that includes unusual characters, represented by codes outside the normal ASCII range. Our program could output some very strange results under these circumstances. Suppose the ASCII code 10₁₆ had been adopted to represent a special character. Then, our attempt to look up the index table would load into the Accumulator whatever happened to be in memory byte n - 10₁₆.

There is no telling what could be in this memory byte; in all probability, this byte will be used to store an instruction code, perhaps a two-hexadecimal digit value. Suppose it contained 2A₁₆; the next program step will double 2A₁₆, add it to the base address of the delay table and access the initial delay code from memory location m + 54₁₆.

Given the microcomputer configuration illustrated in Figure 4-2, this memory location could easily be one of the duplicate addresses which spuriously access some memory byte, because we have used disarmingly simple chip select logic. Had we used more complex chip select logic, then chances are we would now be attempting to access a memory byte that did not exist. In the former case, there is no telling what length of hammer pulse would be generated; in the latter case, an extremely long hammer pulse would be generated, since we would retrieve 0 from a non-existent memory location, and this value would be interpreted as the initial delay constant for the long delay program loop. The hammer pulse would be 720 milliseconds long:



Now in order to avoid this problem we have two options:

- 1) Program logic can simply ignore any invalid ASCII code.
- 2) Program logic can generate a default hammer pulse width for invalid ASCII codes.

If we ignore special characters, the conclusion is obvious: the microcomputer system cannot be used in any application that requires special characters to be printed. Since the special character is ignored, nothing will happen when such a character code is detected on input — there will be no hammer pulse, no carriage movement and no positioning.

Providing a default hammer pulse for special characters means that such characters will be printed, but they may create unevenness in the density of the typed text.

You, as the logic designer, would have to specify your preference.

Either instruction sequence may be inserted into the existing program as follows:

Check for valid ASCII codes inserted here	→	OUT	2	
		IN	1	:INPUT ASCII CHARACTER TO ACCUMULATOR
		ANI	7FH	:MASK OUT HIGH ORDER BIT
		SUI	20H	:SUBTRACT 20H
		LXI	H,INDX	:LOAD INDEX TABLE BASE ADDRESS TO HL
		ADD	L	:ADD ACCUMULATOR CONTENTS TO HL
		MOV	L,A	
		MOV	A,M	:LOAD INDEX INTO ACCUMULATOR
		ADD	A	:MULTIPLY BY 2

Here is the instruction sequence which ignores non-standard ASCII codes:

```

-
-
IN      1      ;INPUT ASCII CHARACTER TO ACCUMULATOR
ANI     7FH    ;MASK OUT HIGH ORDER BIT
COMPARE ASCII CODE WITH LOWEST LEGAL VALUE
CPI     20H
JM      PRD    ;IF CODE IS 1FH OR LESS, BYPASS HAMMER FIRING
;COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE
CPI     7AH
JP      PRD    ;IF CODE IS 7BH OR GREATER, BYPASS HAMMER FIRING
;ASCII CODE IS VALID
SUI     20H    ;SUBTRACT 20H
LXI     H,INDEX ;LOAD INDEX TABLE BASE ADDRESS TO HL
-
-

```

The second option, illustrated below, prints unknown characters with a median density, using density code 3:

```

-
-
IN      1      ;INPUT ASCII CHARACTER TO ACCUMULATOR
ANI     7FH    ;MASK OUT HIGH ORDER BIT
;COMPARE ASCII CODE WITH SMALLEST LEGAL VALUE
CPI     1FH
JP      OK     ;IF CODE IS 20H OR MORE, TEST FOR HIGH LIMIT
;CODE IS ILLEGAL. ASSUME A DENSITY OF 3
NOK     MVI     A,6 ;LOAD TWICE DENSITY
        JMP     NEXT
;COMPARE ASCII CODE WITH LARGEST LEGAL VALUE
OK       CPI     7AH
        JP      NOK ;IF CODE IS 7BH OR GREATER, ASSUME DENSITY OF 3
;ASCII CODE IS VALID
SUI     20H    ;SUBTRACT 20H
LXI     H,INDEX ;LOAD INDEX TABLE BASE ADDRESS INTO HL
ADD     L      ;ADD ACCUMULATOR CONTENTS TO HL
MOV     L,A
MOV     A,M    ;LOAD INDEX INTO ACCUMULATOR
ADD     A      ;MULTIPLY BY 2
NEXT     LXI     H,DELY ;LOAD DELAY TABLE BASE ADDRESS INTO HL
-
-

```

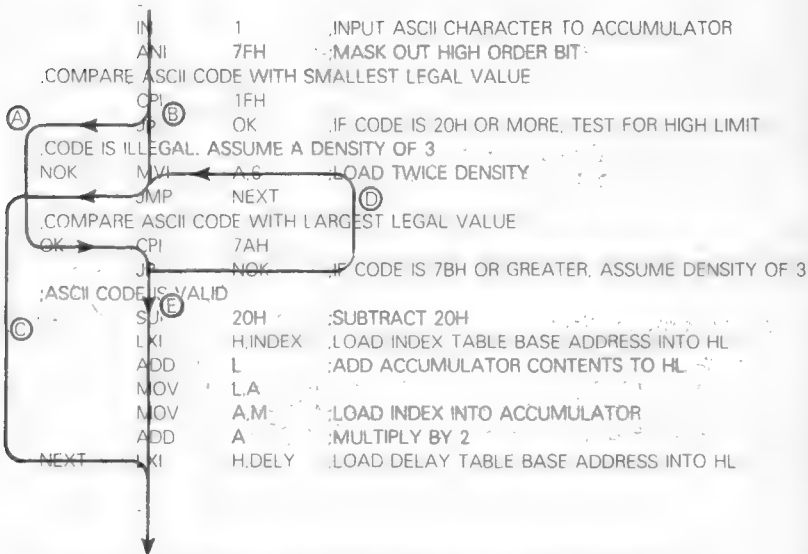
Both of the invalid ASCII code instruction sequences are simplistic in their solution to the problem.

The only new feature introduced is the use of the Compare Immediate (CPI) instruction. This instruction subtracts the immediate data in the operand from the contents of the Accumulator. The result of the subtraction is discarded, which means that the Accumulator contents are not altered; however, status flags are set to reflect the results of the subtraction. We use a JM (Jump Minus) instruction to identify a negative result, which means that the immediate data in the operand was larger than the value in the Accumula-

COMPARE IMMEDIATE
JUMP ON CONDITION

In the second instruction sequence, if the value in the immediate operand is less than, or equal to the contents of the Accumulator, the JP instruction causes a branch to a later instruction labeled OK. The actual program execution paths for the second instruction sequence may appear a trifle confusing to you if you are new to programming, we therefore **illustrate execution paths as follows:**

CONDITIONAL INSTRUCTION EXECUTION PATHS



(A) An ASCII code passes the "lowest legal value" test, but now must be tested for the "highest legal value".

(B) The ASCII code failed the "lowest legal value" test. The program loads twice the default density into the Accumulator and branches to the instruction sequence which accesses the delay constant appropriate to this default density. This Jump is illustrated by (C).

(C) A character which has passed the "lowest legal ASCII value" test is next checked for "highest legal ASCII value". If it fails this test then program execution branches, as shown by (D), to instructions which assume a default density of 3. (D), in fact, meets (B).

(D) An ASCII character that passes both the "lowest legal value" test and the "highest legal value" test is processed via instruction path (E). Instructions in this path load the appropriate density index into the Accumulator.

RESET AND INITIALIZATION

Reset instructions will be executed whenever RESET is input true to the microcomputer system. Initialization instructions will be executed whenever the system is started up.

There is no reason why Reset and Initialization instruction sequences should coincide; in many applications two separate and distinct instruction sequences may be needed. **On the other hand, it is quite common to use Reset in lieu of system initialization.** This means that when you first power up the system, RESET is pulsed true, and this starts the entire microcomputer-based logic system.

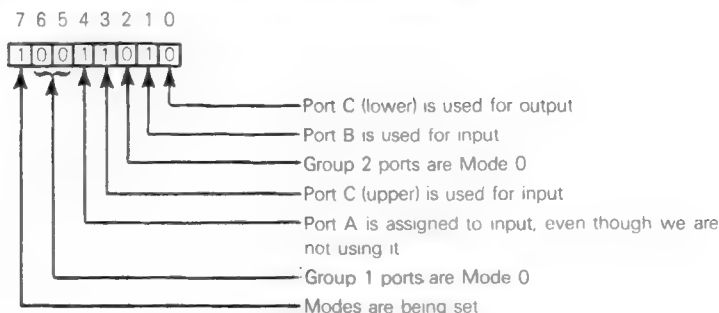
In our case the Reset program is indeed simple. All we have to do is output a control command to I/O Port 3, the control port of the 8255 Programmable Peripheral Interface, then set output signals to the "in between print cycles" condition. The control command selects Mode 0, with appropriate I/O port assignments. **Here is the necessary initialization instruction sequence:**

```

                ORG      0
;SYSTEM RESET AND INITIALIZATION
;OUTPUT CONTROL CODE TO 8255 PROGRAMMABLE
;PERIPHERAL INTERFACE
                MVI      A17
                OUT      3
;SET HAMMER PULSE, PW READY AND
;PW REL HIGH. SET START RIBBON MOTION LOW
                MVI      9AH
                OUT      2

```

This is how the 8255 PPI control code is constructed:



A PROGRAM SUMMARY

First of all, it would be a good idea to put together the entire program, as developed in this chapter. We will include the necessary Assembler directives. This final program is illustrated in Figure 4-6.

Now that the program is finished, notice that RAM memory has not been used. The CPU registers have provided sufficient read/write memory to handle all variable data.

The 1K bytes of ROM program memory are sufficient to contain the entire program, plus the two data tables.

Were you implementing a microcomputer system within the limited confines of the logic included in this chapter, you could now eliminate the two RAM memory chips. In all probability, there would be numerous other logic functions more economically included within the microcomputer system, and these would almost certainly require the presence of some RAM memory. There are nine bytes of read/write memory provided by the seven CPU registers and the CPU Stack Pointer; these are usually insufficient for any real application.

```

INDEX    EQU    0380H    ;EQUATE INDEX TABLE BASE ADDRESS
DELY     EQU    03EEH    ;EQUATE DELAY TABLE BASE ADDRESS - 2
        ORG     0

;SYSTEM RESET AND INITIALIZATION
;INITIALLY SET HAMMER PULSE, PW READY AND
;PW REL HIGH SET START RIBBON MOTION LOW
        MVI     A7
        OUT     2

;OUTPUT CONTROL CODE TO 8255 PROGRAMMABLE
;PERIPHERAL INTERFACE
        MVI     9AH
        OUT     3

;PRINT CYCLE PROGRAM
IN BETWEEN PRINT CYCLES TEST FFI (BIT 5 OF I/O
;PORT 2 FOR A 0 VALUE)
START    IN      2        ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     20H        ;ISOLATE BIT 5
        JNZ     START    ;IF NOT 0, RETURN TO START

;INITIALIZE PRINT CYCLE OUTPUT 0 TO BITS 0
;AND 1 OF I/O PORT 2. OUTPUT 1 TO BIT 3 OF
;I/O PORT 2
        MVI     A,0CH    ;LOAD MASK INTO ACCUMULATOR
        OUT     2        ;OUTPUT TO I/O PORT 2

;OUTPUT 0 TO BIT 3 OF I/O PORT 2 THIS COMPLETES
;START RIBBON MOTION PULSE
        MVI     A,4      ;LOAD MASK INTO ACCUMULATOR
        OUT     2        ;OUTPUT TO I/O PORT 2

;TEST FOR END OF PRINTWHEEL POSITIONING
LOP1     IN      2        ;INPUT I/O PORT 2 TO ACCUMULATOR
        ANI     20H        ;ISOLATE BIT 5
        JZ      LOP1     ;IF 0 RETURN TO LOP1

;EXECUTE PRINTWHEEL SETTLING 2 MS DELAY
        MVI     A,0      ;LOAD ACCUMULATOR WITH 0
LOP2     DCR     A        ;DECREMENT A
        JNZ     LOP2     ;IF A DOES NOT DECREMENT TO 0, RE-DECREMENT

;TEST PRINTHAMMER FIRING CONDITIONS
LOP3     IN      2        ;INPUT I/O PORT 2 TO ACCUMULATOR
        RLC          ;MOVE BIT 7 INTO CARRY
        JNC     PRD      ;IF CARRY IS 0, BYPASS PRINTHAMMER FIRING
        ANI     20H        ;ISOLATE BIT 4 WHICH IS NOW BIT 5
        JZ      LOP3     ;WAIT FOR NONZERO VALUE BEFORE FIRING

;FIRE PRINTHAMMER
        IN      2        ;SET HAMMER PULSE LOW. OUTPUT 0
        ANI     FBH        ;TO BIT 2 OF I/O PORT 2
        OUT     2
        IN      1        ;INPUT ASCII CHARACTER TO ACCUMULATOR
        ANI     7FH        ;MASK OUT HIGH ORDER BIT

;COMPARE ASCII CODE WITH LOWEST LEGAL VALUE
        CPI     20H
        JM      PRD      ;IF CODE IS 1FH OR LESS, BYPASS HAMMER FIRING

;COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE
        CPI     7AH
        JP      PRD      ;IF CODE IS 7BH OR GREATER, BYPASS HAMMER FIRING

```

Figure 4-6. A Simple Print Cycle Program

ASCII CODE IS VALID

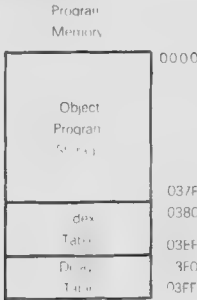
```

    SUI    20H    ;SUBTRACT 20H
    LXI    H,INDEX ;LOAD INDEX TABLE BASE ADDRESS TO HL
    ADD    L      ;ADD ACCUMULATOR CONTENTS TO HL
    MOV    L,A
    MOV    A,M    ;LOAD INDEX INTO ACCUMULATOR
    ADD    A      ;MULTIPLY BY 2
    LXI    H,DELY ;LOAD DELAY TABLE BASE ADDRESS INTO HL
    ADD    L      ;ADD ACCUMULATOR CONTENTS TO HL
    MOV    L,A
    MOV    E,M    ;LOAD DELAY CONSTANT INTO D,E
    INX    H
    MOV    D,M
LOP4    DCX    D    ;EXECUTE LONG DELAY
    MOV    A,D
    ORA    E
    JNZ    LOP4
    IN     2      ;AT END OF DELAY OUTPUT 1 TO BIT 2
    ORI    4      ;OF I/O PORT.2 (HAMMER PULSE HIGH)
    OUT    2
;EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
    LXI    D,F7H  ;LOAD TIME CONSTANT INTO D,E
LOP5    DCR    D    ;DECREMENT D,E
    MOV    A,D    ;TEST FOR 0 IN D,E
    ORA    E
    JNZ    LOP5    ;REDECREMENT IF NOT 0
                    ;OUTPUT 1 TO BIT 0 OF I/O PORT 2. THIS SETS
                    ;PW REL HIGH
    IN     2      ;INPUT I/O PORT 2 TO ACCUMULATOR
    ORI    1      ;SET BIT 0 TO 1
    OUT    2      ;OUTPUT RESULT
;EXECUTE A 2 MILLISECOND PRINTWHEEL READY DELAY
PRD    MVI    A,0  ;LOAD ACCUMULATOR WITH 0
LOP6    DCR    A    ;DECREMENT A
    JNZ    LOP6    ;IF A DOES NOT DECREMENT TO 0, RE-DECREMENT
                    ;TEST FOR EOR DET (BIT 6 OF I/O PORT 2) EQUAL
                    ;TO 0 AS A PREREQUISITE FOR ENDING THE PRINT CYCLE
LOP7    IN     2      ;INPUT I/O PORT 2 TO ACCUMULATOR
    ANI    40H    ;ISOLATE BIT 6
    JZ     LOP7    ;RETURN AND RETEST IF NOT 0
                    ;AT END OF PRINT CYCLE SET BIT 1 OF I/O PORT 2 TO 1
                    ;THIS SETS CH RDY HIGH
    IN     2      ;INPUT I/O PORT 2 TO ACCUMULATOR
    ORI    2      ;SET BIT 1 TO 1
    OUT    2      ;OUTPUT RESULT
    JMP    START  ;JUMP TO NEW PRINT CYCLE TEST
;INDEX TABLE FOLLOWS HERE
    ORG    0380H
    Data represented by 90 index entries follow here. Data appears in mnemonic field,
    one byte per line.
;DELAYS TABLE FOLLOWS HERE
    ORG    03F0H
    Data representing 6 delays follow here. Data appears in mnemonic field, two bytes
    per line.

```

Figure 4-6. A Simple Print Cycle Program (Continued)

Here is the final program memory map identifying the way in which the program illustrated in Figure 4-6 uses ROM memory:



Chapter 5

A PROGRAMMER'S PERSPECTIVE

The program we developed in Chapter 4 is considerably shorter and easier to follow than the digital simulation of Chapter 3. While we came a long way in Chapter 4 we still have a way to go. The program in Figure 4-6 treats the logic to be implemented as a single transfer function, but it is not a well written program.

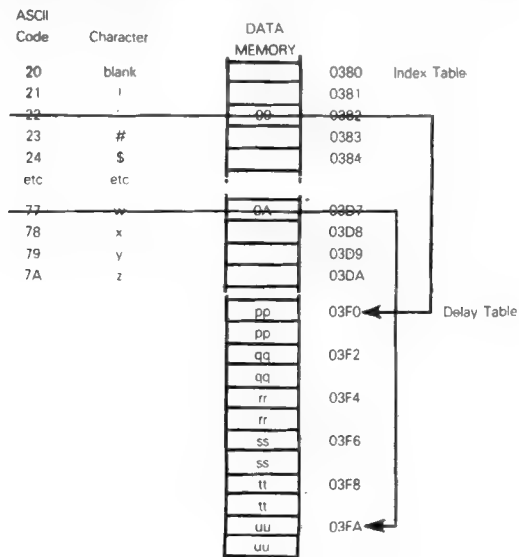
To the digital logic designer, one of the most confusing things about programming is the trivial ease with which you can do the same thing in ten different ways. Does this imply that some implementations are more efficient than others? Indeed yes. To a great extent writing efficient programs is a talent, just as creating efficient digital logic is a talent; but there are certain rules which, if followed, will at least help you avoid obvious mistakes. In this chapter we are going to take the program created in Chapter 4 and look at it a little more carefully.

SIMPLE PROGRAMMING EFFICIENCY

The first thing you should do, after writing a source program, is to go back over it, looking for elementary ways in which you can cut out instructions.

EFFICIENT TABLE LOOKUPS

On average, you will find that it is possible to reduce a program to two-thirds of its original length, simply by writing more efficient instruction sequences. In Figure 4-6, the most obvious example of sloppy programming involves the Index Table. The program loads a value between 1 and 6 from an Index Table byte, then multiplies this value by two before adding it to the base address of the Delay Table. **Why not directly store twice the index in the Index Table?** That cuts out one instruction as follows



:ASCII CODE IS VALID

```

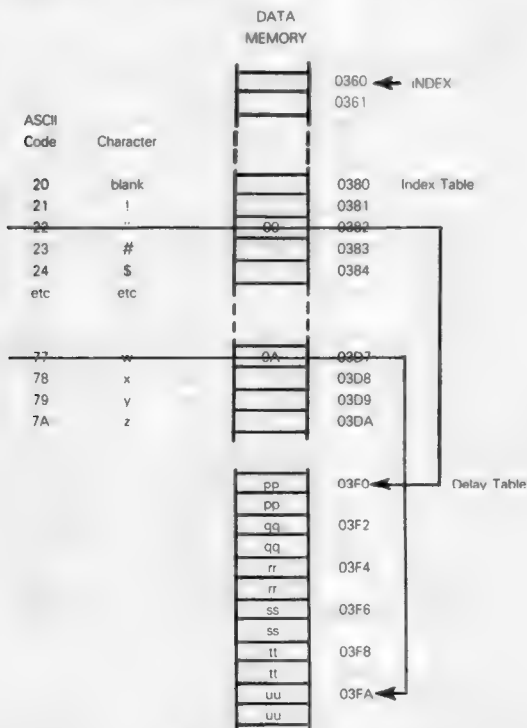
SUI    20H    :SUBTRACT 20H
LXI    H,INDEX :LOAD INDEX TABLE BASE ADDRESS
ADD    L,HL   :ADD ACCUMULATOR CONTENTS TO HL
MOV    L,A    :
MOV    A,M    :LOAD INDEX X2 INTO ACCUMULATOR
LXI    H,DELY :LOAD DELAY TABLE BASE ADDRESS INTO HL
ADD    L,HL   :ADD ACCUMULATOR CONTENTS TO HL
MOV    L,A    :

```

ADD
instruction
dropped

In the instruction sequence above, notice that one instruction has been removed following the shaded MOV instruction.

There are still a number of additional ways in which we can make the Delay Table lookup more efficient. **Why subtract 20₁₆ from the ASCII code**, for example? If we are going to add the ASCII code to a base address, there is nothing to stop us Equating this base address, represented by the symbol INDEX, to a value 20₁₆ less than the first real Index Table byte. Our instruction sequence now collapses further, as follows:



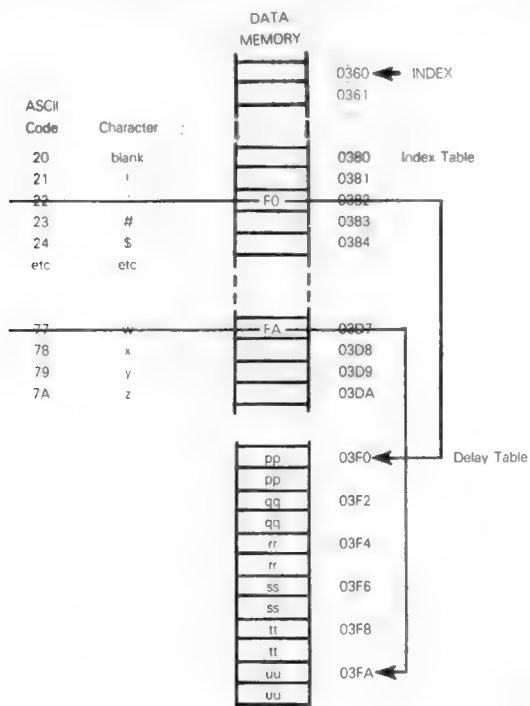
```
INDEX EQU 0360H ;EQUATE INDEX TO TABLE BASE ADDRESS 20H
```

```

:ASCII CODE IS VALID
SUI      LXI H,INDEX ;LOAD INDEX TABLE BASE ADDRESS -20H
instruction ADD L ;ADD ACCUMULATOR CONTENTS TO HL
dropped  MOV L,A
          MOV A,M ;LOAD INDEX X2 INTO ACCUMULATOR
          LXI H,DELY ;LOAD DELAY TABLE BASE ADDRESS INTO HL
          ADD L ;ADD ACCUMULATOR CONTENTS TO HL
          MOV L,A

```

Okay, so INDEX is now being equated to 0360₁₆ which means that we no longer need to subtract 20₁₆ from the ASCII code. We have eliminated the SUI instruction which was above the shaded LXI instruction. **Now instead of storing twice the character density index in the Index Table, why not store the second half of the Delay Table address?** Our program will now contract further as follows:



```
INDEX EQU 0360H ;EQUATE INDEX TO TABLE BASE ADDRESS 20H
```

```

:ASCII CODE IS VALID
LXI H,INDEX ;LOAD INDEX TABLE BASE ADDRESS -20H
ADD L ;ADD ACCUMULATOR CONTENTS TO HL

```



```

MOV  R, L,A
MOV  R, L,M      ;LOAD LOW ORDER BYTE OF DELAY TABLE ADDRESS
MVI  R, H,03H    ;LOAD HIGH ORDER BYTE OF DELAY TABLE ADDRESS

```

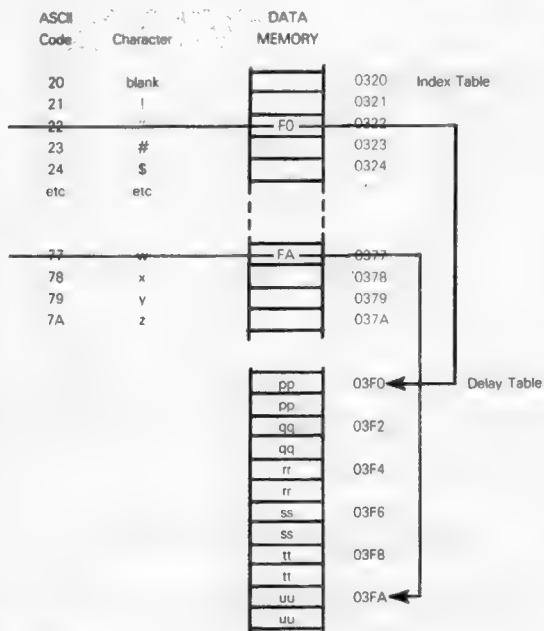
Two more instructions have disappeared.

We have now taken out four instructions from the sequence which loads the printhammer firing initial delay constant — and we are still not done.

Why not move the whole Index Table, so that instead of occupying memory locations 0380₁₆ through 03DA₁₆, it occupies memory locations 0320₁₆ through 037A₁₆?

The ASCII code, stripped of the parity bit, now becomes the low order byte of the Index Table address; and our instruction sequence contracts further as follows:

**TABLES POSITIONED
TO SIMPLIFY
ACCESS
INSTRUCTION
SEQUENCE**



:ASCII CODE IS VALID

```

MVI  R, H,03H    ;LOAD INDEX TABLE ADDRESS, HIGH ORDER BYTE
MOV  R, L,A      ;MOVE LOW ORDER BYTE OF ADDRESS TO L
MOV  R, L,M      ;LOAD LOW ORDER BYTE OF DELAY TABLE ADDRESS

```

Suppose a "w" character is to be printed. Before the first of the above three instructions is executed, the Accumulator contains 77₁₆, as a result of the previous:

```

IN  R, 1
ANI R, 7FH

```

instructions' execution. Following execution of the:

```

MVI  R, H,03H

```

instruction, the H register will contain 03_{16} ; this is the upper half of the implied memory address. Next the instruction:

MOV L,A

moves 77_{16} from the Accumulator to the L register. H and L now contain 0377_{16} , this is the effective implied address. The next instruction:

MOV L,M

moves, to the L register, the contents of the memory byte addressed by HL

HL contains 0377_{16} . Memory byte 0377_{16} contains FA_{16} , therefore FA_{16} is moved to the L register. The new implied address is $03FA_{16}$, and that is the required Delay Table address.

Nine instructions have been reduced to three and the only price paid is that we have had to move the Index Table to a new area of data memory.

To insure that you understand how the program will now look, the old and new instruction sequences are shown side-by-side below, without comment fields:

Old Program		New Program	
ASCII CODE IS VALID			
SUI	20H	MVI	H,03H
LXI	H,INDEX	MOV	L,A
ADD	L	MOV	L,M
MOV	L,A		
MOV	A,M		
ADD	A		
LXI	H,DELY		
ADD	L		
MOV	L,A		

Unfortunately there are no golden rules which, if followed, will ensure that you always write the shortest program possible. Once you have written a few programs, you will understand how individual instructions work, and that, in turn, generates efficiency. The purpose of the preceding pages has been to demonstrate the enormous difference between a compact program and a straightforward program. If your product is going to be produced in high volume, it behooves you to spend the time and money cutting down program size — then you may be able to eliminate some of your ROM chips.

HARDWARE UTILIZATION

All computer programmers try to write efficient assembly language programs. However, only microcomputer programmers must consider hardware utilization as an integral contributor to programming efficiency.

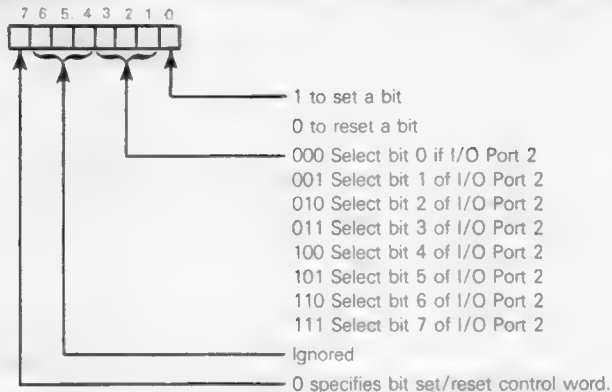
Now we used the 8255 Programmable Peripheral Interface in Mode 0, accessing port bits in the most obvious way. Let us explore some of the alternatives.

HARDWARE-SPECIFIC INSTRUCTIONS

Observe that much of the time we are setting and resetting individual bits which become input and output signals. The 8255 PPI has a Control I/O port, which in our case is

**BIT SET/RESET
INSTRUCTIONS**

addressed as I/O Port 3. Individual bits of I/O Port 2 can be set or reset by outputting an appropriate control word to I/O Port 3. This is the format of the control word:



Notice that this bit set/reset instruction is hardware dependent, it relies on logic within the 8255 PPI for the requirements of the instruction to be implemented.

**HARDWARE
DEPENDENT
INSTRUCTIONS**

The bit set/reset control instructions do not save steps when reading the status of an input signal; but they do convert three instructions into two instructions whenever we want to change the status of an output signal. Going through the program summarized in Figure 4-6, here are the effective changes:

Old Program New Program

```

;FIRE PRINTHAMMER. SET HAMMER PULSE LOW.
;OUTPUT 0 TO BIT 2 OF I/O PORT 2.
IN      2          MVI      A,4
ANI     FBH        OUT     3
OUT     2
.
.
;AT END OF DELAY OUTPUT 1 TO BIT 2 OF I/O PORT 2
;(HAMMER PULSE HIGH)
IN      2          MVI      A,5
ORI     4          OUT     3
OUT     2
.
.
;OUTPUT 1 TO BIT 0 OF I/O PORT 2. THIS SETS
;PW REL HIGH
IN      2          MVI      A,1
ORI     1          OUT     3
OUT     2
.
.
;AT END OF PRINT CYCLE SET BIT 1 OF I/O PORT 2 TO 1.
;THIS SETS CH RDY HIGH
IN      2          MVI      A,3
ORI     2          OUT     3
OUT     2          JMP     START
JMP     START

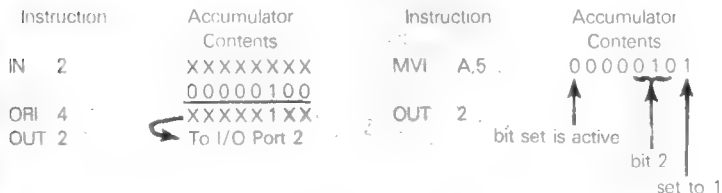
```

In case you still have lingering doubts as to how the bit set/reset works, we will graphically illustrate HAMMER PULSE being output high. This requires 1 to be output to bit 2 of I/O Port 2:

BIT SET/RESET ILLUSTRATED

Old Program

New Program

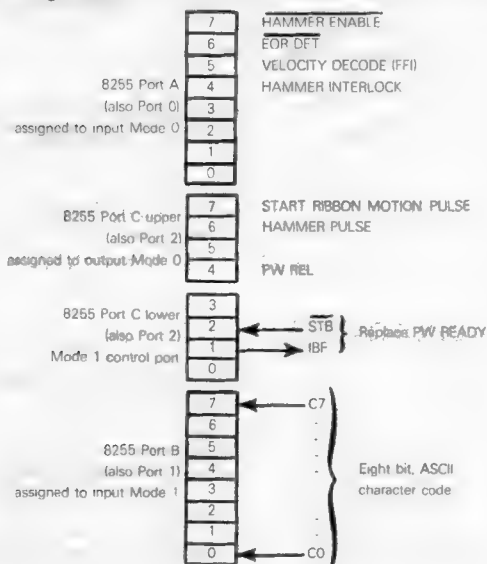


DIRECT USE OF HARDWARE FEATURES

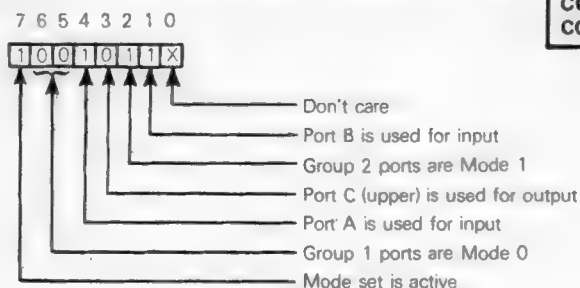
We are going to assume that external logic uses the PRINTWHEEL READY signal to ensure that it does not attempt to input a new character code until the prior character code has been processed. We expend some instructions setting PRINTWHEEL READY low at the beginning of the print cycle, then resetting it/high at the end of the print cycle.

Without a clear definition of the logic external to our microcomputer system, we have no way of knowing whether the PRINTWHEEL RELEASE and PRINTWHEEL READY output signals are needed externally to define specific time delays, or whether they are simply being used to ensure that we allow one character to complete printing before trying to start printing the next character. If the only function of the PRINTWHEEL READY signal is to ensure that a new character is not input to the microcomputer system before the old character has been printed, then we can dispense with the PRINTWHEEL READY signal and replace it with

strobed input, using Mode 1 for I/O Port 1. This is how our I/O ports will now be assigned:

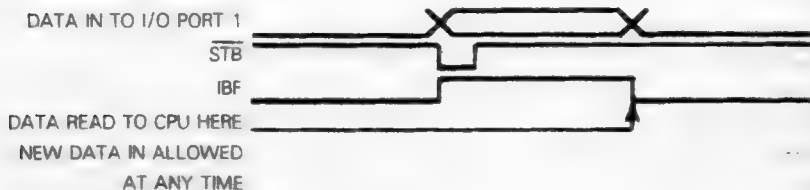


This is how the 8255 PPI control code is constructed:



**PPI
CONTROL
CODE**

When external logic inputs a new character at I/O Port 1, it simultaneously inputs a low strobe signal at pin 2 of I/O Port 2 (STB). At this instant the 8255 PPI will output IBF high at pin 1 of I/O Port 2. IBF will remain high until instructions are executed to read the contents of I/O Port 1 into the CPU. At this time IBF will be reset low. This may be illustrated as follows:



Now the print cycle logic only reads the contents of I/O Port 1 once during any print cycle. In Mode 1, I/O Port 1 is buffered, therefore all external logic has to do is use IBF as a strobe signal. So long as IBF is low, external logic is free to input the next ASCII character. If IBF is high, then external logic must wait.

This scheme eliminates the PRINTWHEEL-READY signal and any instructions needed to manipulate this signal.

Notice that the data signals which were output via bits 0 through 3 of I/O Port 2 have been moved to bits 7 through 4 of I/O Port 2. The signals that were allocated to bits 7 through 4 of I/O Port 2 have been moved to I/O Port 0. There is good reason for this movement. Remember, the bit set/reset controls only works for I/O Port 2. If we want to reduce from 3 to 2 the number of instructions required to change the status of an output signal, then the output signals must be assigned to pins of I/O Port 2. Since we cannot reduce the number of instructions required to sample input signals, we might as well move these signals to I/O Port 0.

SUBROUTINES

If you look again at the program in Figure 4-6, you will notice that at two points within this program we execute identical instruction sequences to create a 2 millisecond delay. Now it only takes three instructions to execute a 2 millisecond delay, so the fact that these three instructions have been repeated is no big tragedy. If you think about it, however, the potential exists for some very uneconomical memory utilization in longer programs.

We have kept our program simple in Chapter 4 because it must remain small enough to handle in a book; but project, if you will, a more complex routine where a 30 instruction sequence needs to be repeated, rather than a 3 instruction sequence. We must now find some way of including the instruction sequence just once, then branching to this single sequence from a number of different locations within a program, as needed. That is what a subroutine will do for you.

Let us take the 3 instructions which execute a 2 millisecond delay and convert them into a subroutine. This is what happens to relevant portions of the program:

```

      ORG      0
      LXI      H,08FFH      ;INITIALIZE STACK POINTER TO END OF
      SPHL                      ;DATA AREA
      -
      -

;EXECUTE PRINTWHEEL SETTling 2 MS DELAY
      CALL     D2MS
      -
      -

;EXECUTE A 2-MILLISECOND PRINTWHEEL READY DELAY
PRD      CALL     D2MS
      -
      -

;AT END OF PRINT CYCLE SET BIT 1 OF I/O PORT 2 TO 1
;THIS SETS CH RDY HIGH
      MVI      A,3          ;THESE ARE THE NEW INSTRUCTIONS
      OUT      3            ;TO SET A BIT OF I/O PORT 2
      JMP      START

```

;SUBROUTINE TO EXECUTE A 2 MILLISECOND DELAY

D2MS MVI A,0 ;LOAD ACCUMULATOR WITH 0

LOPD DCR A ;DECREMENT A

JNZ LOPD ;IF A DOES NOT DECREMENT TO 0, RE-DECREMENT

RET ;RETURN FROM SUBROUTINE

In order to understand how a subroutine works, we will assign some arbitrary memory addresses for our source program's object code. we will show, step-by-step, what happens when a subroutine is called and what happens upon returning from the subroutine. First of all, **here is the assumed memory map:**



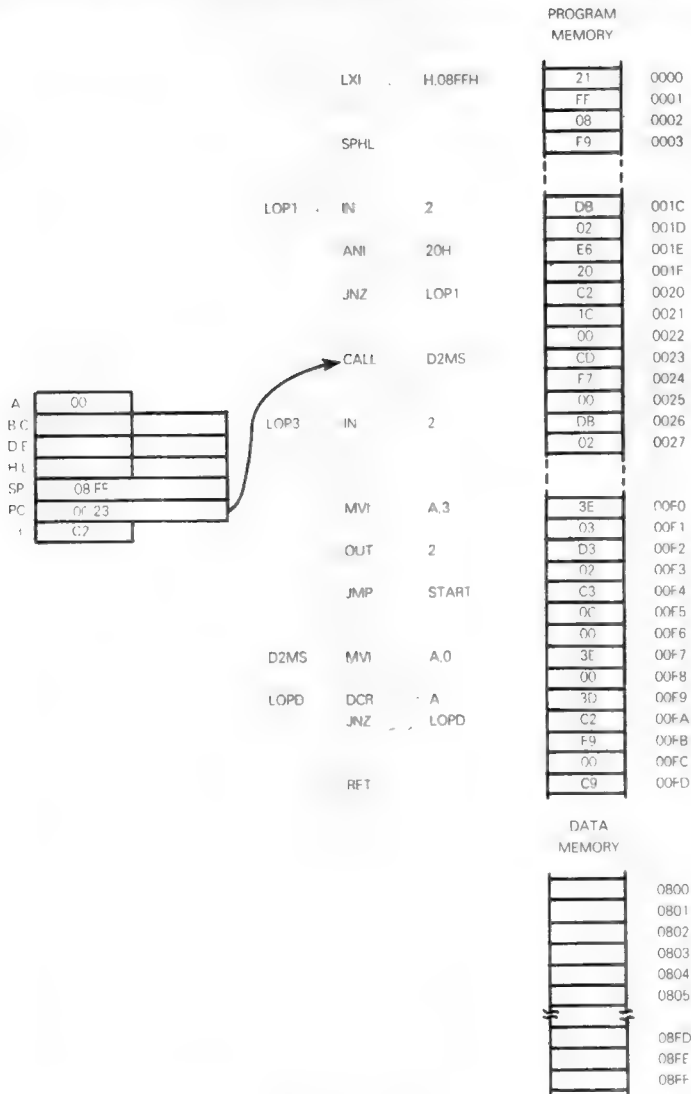
		PROGRAM MEMORY	
LXI	H,08FFH	21	0000
		FF	0001
		08	0002
SPHL		F9	0003
LOP1	IN 2	0B	001C
		02	001D
ANI	20H	E6	001E
		20	001F
JNZ	LOP1	C2	0020
		1C	0021
		00	0022
CALL	D2MS	CD	0023
		F7	0024
		00	0025
LOP3	JN 2	0B	0026
		02	0027
	MVI A,3	3E	00F0
		03	00F1
	OUT 2	D3	00F2
		02	00F3
	JMP START	C3	00F4
		0C	00F5
		00	00F6
D2MS	MVI A,0	3E	00F7
		00	00F8
LOPD	DCR A	3D	00F9
	JNZ LOPD	C2	00FA
		F9	00FB
		00	00FC
	RET	C9	00FD

DATA MEMORY



SUBROUTINE CALL

Suppose we are about to execute the first CALL D2MS instruction. At this point registers will contain the following data:



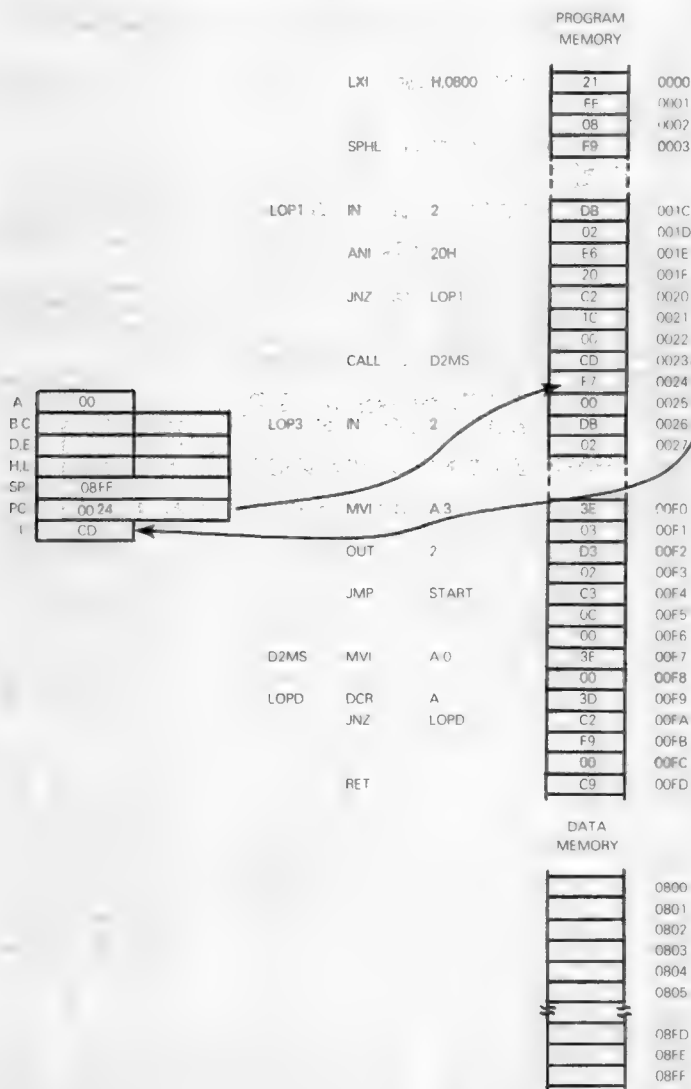
The Program Counter (PC) addresses the first byte of the Call instruction's object code; this address is 0023_{16} . The instruction register holds the object code for the most recently executed instruction; this is a JNZ instruction located at byte 0020_{16} . The Stack Pointer, you will notice, was initialized at the beginning of the program; it contains $08FF_{16}$. According to Figure 4-2, this is the

address of the first byte of read/write memory. Since the stack has not been used, the stack pointer will still contain 08FF₁₆.

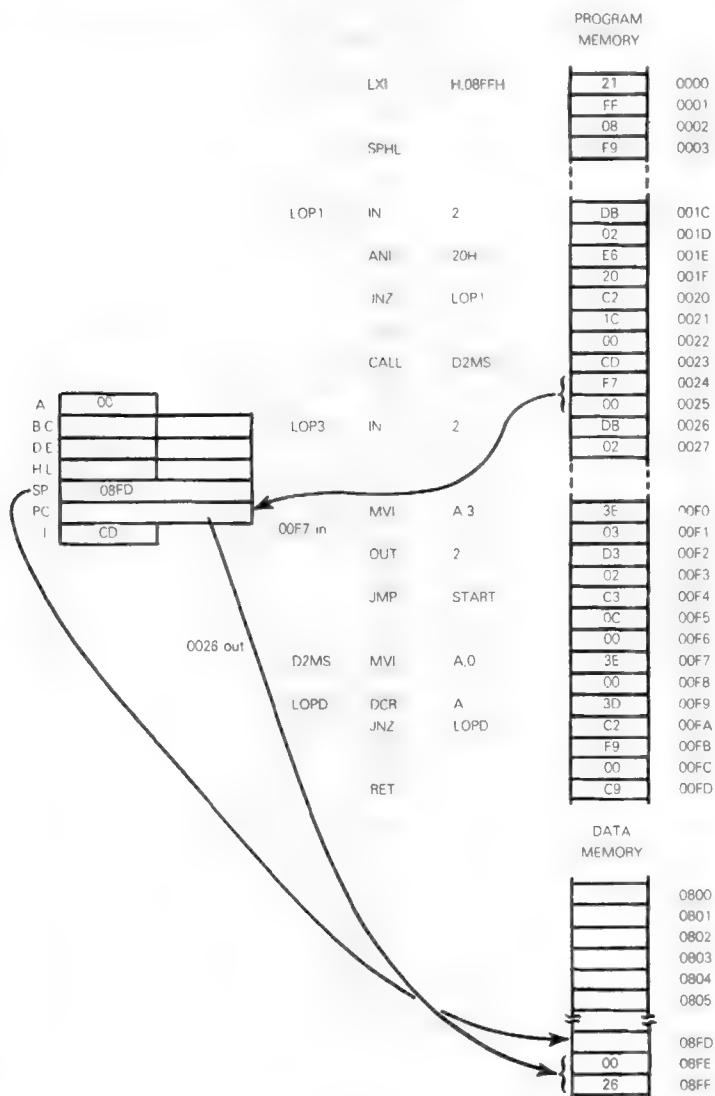
The Accumulator contains 00 because this was the condition which caused execution to break out of the holding loop starting at LOP1.

Now when the Call instruction is executed, steps occur as follows:

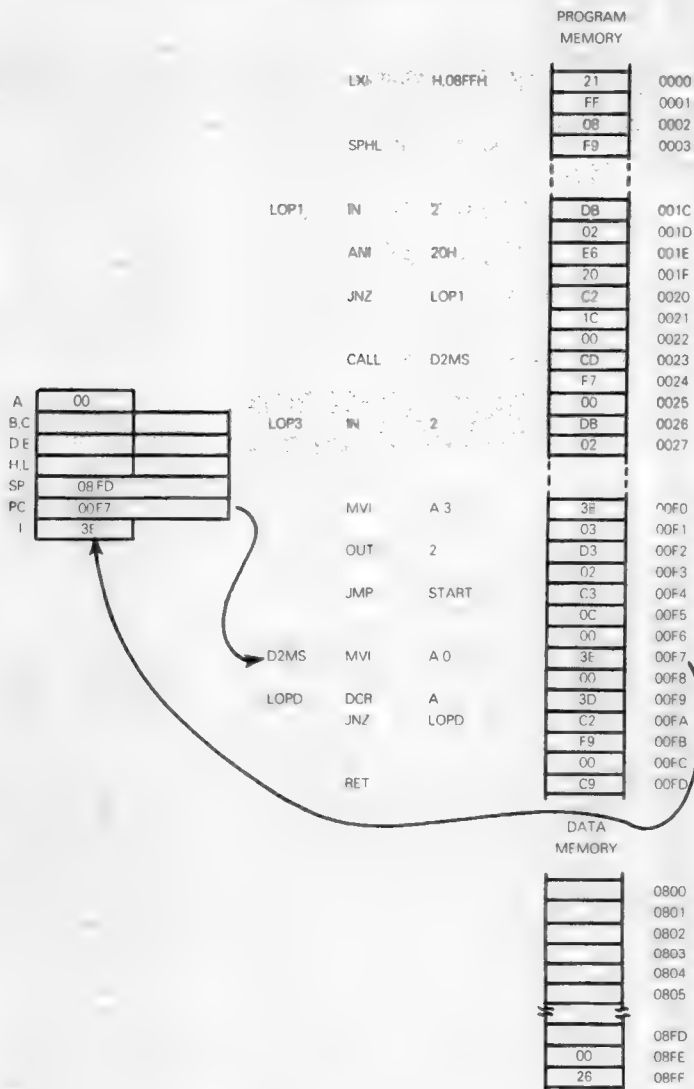
The Call instruction object code is loaded into the Instruction register and the Program Counter is incremented:



The Program Counter is incremented by 2 to bypass the CALL address. This incremented value is saved in the first two stack bytes. The CALL address is then loaded into the Program Counter. The Stack Pointer is decremented by 2 so that it addresses the first free stack byte:



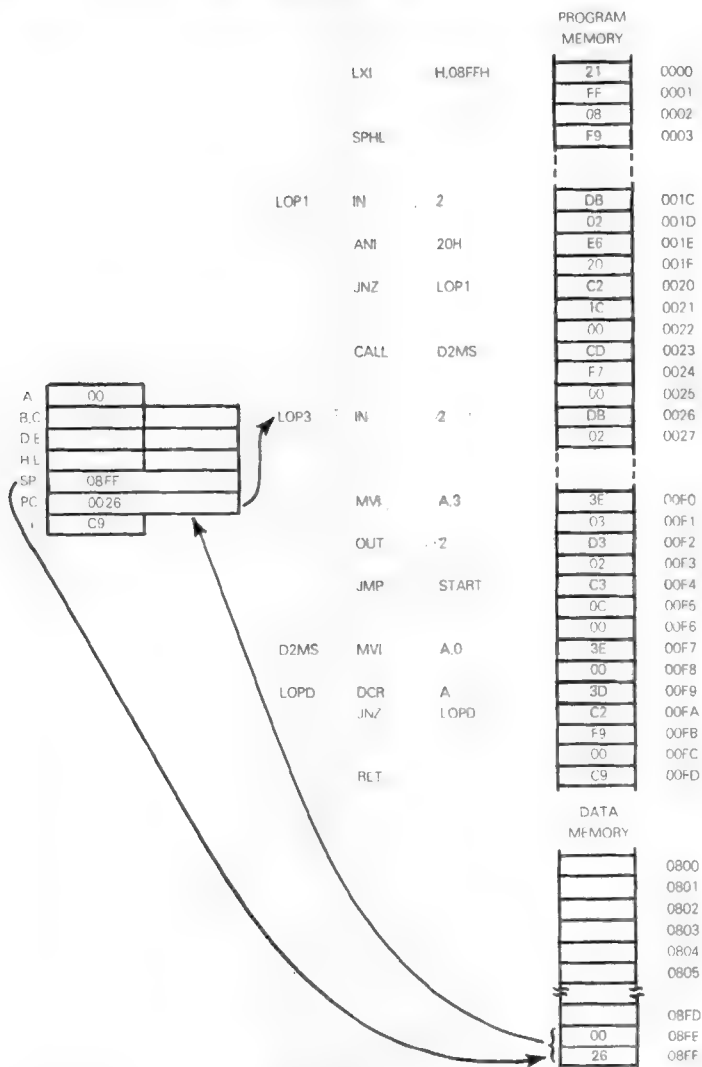
The next instruction executed has its object code stored in memory byte 00F7₁₆, this is the memory byte now addressed by the Program Counter:



Instructions within the 2 millisecond delay loop are now executed repetitively until the Accumulator contents decrements from 01 to 00. Remember, the first time the Accumulator is decremented it will go from 00 to FF₁₆ and that is why the two instruction loop beginning at LOPD will be executed 256 times.

SUBROUTINE RETURN

When the Accumulator finally decrements from 01 to 00, execution passes to the Return (RET) instruction. This instruction increments the contents of the Stack Pointer by 2, then moves the contents of the two top stack bytes into the Program Counter. Thus, program execution returns to the instruction that follows the Call:



In summary, this is what happened:

When the Call instruction was executed, the address of the next instruction was saved in the stack. The Call instruction provided the address of the next instruction to be executed.

The next instruction to be executed was the first instruction of the subroutine.

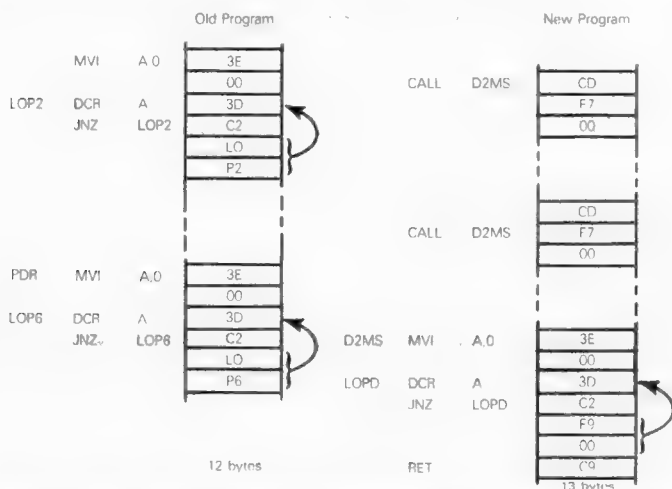
The last instruction of the subroutine merely caused the address saved at the top of the stack to be returned to the Program Counter; and this, in turn, caused execution to branch back to the instruction following the Call.

WHEN TO USE SUBROUTINES

There is a price associated with using subroutines:

- 1) Each Call instruction represents three additional bytes of object code.
- 2) The instruction sequence which has been moved to the subroutine must have an appended Return instruction which costs one byte of object code.

Let us first look at our specific case. The three instructions which constitute the 2 millisecond delay occupy 6 bytes of object code. These three instructions occur twice, therefore, combined, they occupy 12 bytes of object code. When moved to a subroutine, adding the Return instruction increases the object code bytes from 6 to 7. In addition, there are two Call instructions and each requires 3 bytes of object code — which means that the two Call instructions, plus the subroutine, generate 13 bytes of object code. This may be illustrated as follows:



In our specific case, therefore, moving the 2 millisecond delay instruction sequence into a subroutine has cost us one byte of object code. It has cost us four additional bytes of object code — required to initialize the Stack Pointer; and our microcomputer system is now going to require RAM memory.

A stack can only exist if read/write memory is present.

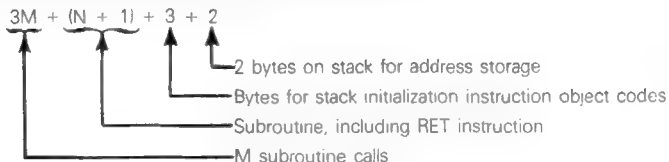
Now these comments do not imply that subroutines are a dubious programming feature, to be used sparingly; on the contrary, it is hard to conceive of any program which, when well written, will not include some subroutines. But bear in mind that **there is a minimum subroutine size below which subroutines in general become uneconomical.**

Suppose there are **N bytes of object code** in an instruction sequence which you are planning to convert into a subroutine.

Suppose **the N bytes of object code occur M times**; that means when the N bytes of object code become a subroutine, it will be called by M CALL instructions.

Without subroutines, M x N bytes will be consumed repeating N bytes M times.

With subroutines, the number of bytes consumed is:



For the subroutine to be worthwhile, $3M + N + 6$ must be less than $M \times N$.

Table 5-1 shows the minimum economic subroutine length as a function of the number of subroutine calls.

Table 5-1. The Shortest Economic Subroutine Length As A Function Of The Number Of Times The Subroutine Is Called

Number Of Subroutine Calls (M)	Minimum Economic Subroutine Length (N)
2	12 Bytes
3	8 Bytes
4	6 Bytes
5	6 Bytes
10	4 Bytes
20	4 Bytes

CONDITIONAL SUBROUTINE RETURNS

Even though none of the repeated instruction sequences within the program in Figure 4-6 are long enough to justify being turned into a subroutine, we will nonetheless explore the potential of subroutines further.

Just as there are conditional Jump instructions, which we use frequently within a time delay loop, so there are conditional subroutine Call instructions and conditional Return from Subroutine instructions.

Conditional subroutine Call and Return instructions are particularly useful in longer subroutines within which there are variable execution paths.

Consider the printhammer firing instruction sequence in Figure 4-6. Given the program as illustrated, this instruction sequence occurs just once, which means that converting it into a subroutine would make no sense. **It is possible to imagine a more extensive program which performs a wide variety of printer interface operations, such that printhammer firing logic might be triggered for a number of different reasons. Since the printhammer firing logic consists of a fairly long set of instructions, putting these instructions in a subroutine would be absolutely mandatory. Consider the following subroutine implementation:**

:PRINTHAMMER FIRING SUBROUTINE

```

PFIR    IN      2          ;INPUT I/O PORT 2 TO ACCUMULATOR
RLC          ;MOVE BIT 7 INTO CARRY
RNC     ;IF CARRY IS NOT SET, RETURN
  
```

```

    ANI    20H    ;ISOLATE BIT 4 WHICH IS NOW BIT 5
    RZ      ;IF ZERO, RETURN
;FIRE PRINTHAMMER
    MVI    A4     ;SET HAMMER PULSE LOW. OUTPUT 0
    OUT    3     ;TO BIT 2 OF I/O PORT 2
    IN     1     ;INPUT ASCII CHARACTER CODE TO ACCUMULATOR
    ANI    7FH    ;MASK OUT HIGH ORDER BIT
;COMPARE ASCII CODE WITH LOWEST LEGAL VALUE
    CPI    20H
    RM      ;IF CODE IS 1FH OR LESS, BYPASS HAMMER FIRING
;COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE
    CPI    7AH
    RP      ;IF CODE IS 7BH OR GREATER, BYPASS FIRING
;ASCII CODE IS VALID
    MVI    H,03H  ;LOAD INDEX TABLE ADDRESS, HIGH ORDER BYTE
    MOV    L,A    ;MOVE LOW ORDER BYTE OF ADDRESS TO L
    MOV    L,M    ;LOAD LOW ORDER BYTE OF DELAY TABLE ADDRESS
    CALL   LDLY   ;CALL LONG DELAY SUBROUTINE
    MVI    A,5    ;AT END OF DELAY OUTPUT 1 TO BIT 2 OF
    OUT    3     ;I/O PORT 2 (HAMMER PULSE HIGH)
;EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY
    LXI    H,MS3
    CALL   LDLY
;OUTPUT 1 TO BIT 0 OF I/O PORT 2. THIS SETS
;PW REL HIGH
    MVI    A,1
    OUT    3
    RET      ;RETURN FROM SUBROUTINE
;LONG DELAY SUBROUTINE. ASSUME H AND L
;ADDRESS THE FIRST OF TWO DATA BYTES WHICH HOLD
;THE INITIAL DELAY CONSTANT
LDLY  MOV    E,M    ;LOAD INITIAL DELAY CONSTANT
      INX    H
      MOV    D,M
LDLP  DCX    D      ;EXECUTE LONG DELAY
      MOV    A,D
      ORA    E
      JNZ    LDLP
      RET      ;RETURN AT END OF LONG DELAY
MS3   OOF7H      ;PRINTWHEEL RELEASE TIME DELAY CONSTANT

```

The subroutine illustrated above only fires the printhammer if all necessary conditions have been met; a quick exit is executed if any firing condition has not been met. The conditional Return instructions are shaded.

**CONDITIONAL
RETURN**

Notice that we have used the more compact instruction sequences both to output single bit data to I/O Port 2 and to identify the correct printhammer firing delay.

Note also that we have added a subroutine within the subroutine. The long delay instruction sequence has been moved to a subroutine, the first instruction of which is labeled LDLY. This is referred to as a "nested subroutine".

**NESTED
SUBROUTINES**

One novel feature of subroutine LDLY is that it requires the initial delay constant to be stored in two bytes of memory, the first of which is addressed by the H and L registers when LDLY is called. Instructions

**SUBROUTINE
PARAMETER**

within subroutine LDLY will actually load the initial delay constant into the D and E registers. The initial delay constant becomes a parameter, which allows one subroutine to implement a complete spectrum of time delays. Subroutine parameters are a very important feature of subroutine use.

The second time subroutine LDLY is called, instead of loading the required initial constant (F7₁₆ into the D and E registers, we load an address represented by the symbol MS3 into the H and L registers. The symbol MS3 will become the address of two data bytes, somewhere in memory within these two data bytes the value 00F7₁₆ must be stored

MULTIPLE SUBROUTINE RETURNS

Subroutine PFIR is not as useful as it could be. There are four conditional returns from this subroutine, each of which is triggered by a different invalid condition. There is also a subroutine return following valid printhammer firing.

How is the calling program to know whether the printhammer was or was not fired after PFIR was called? Testing statuses is not very safe, since we cannot be certain what happens to status conditions during execution of the printhammer firing instructions themselves. You cannot test the Carry status to determine whether the RNC instruction was the conditional return which caused an exit from subroutine PFIR; this is because you cannot tell what happens to the Carry status while the rest of the subroutine executes. For example, the Carry status will be 0 if the RM conditional return instruction causes an exit from subroutine PFIR

Subroutines which contain a large number of conditional error exits, in addition to a standard return, will often contain logic which returns to a number of different instructions in the calling program. Take the case of subroutine PFIR. The instruction sequence which calls this subroutine may appear as follows:

```

RTO      CALL    PFIR      ;CALL PRINTHAMMER FIRING SUBROUTINE
          JMP     RT1       ;RETURN HERE FOR PRINTWHEEL REPOSITIONING
          JMP     RTO       ;RETURN HERE FOR HAMMER INTERLOCK LOW
          JMP     RT2       ;RETURN HERE FOR ASCII CODE LESS THAN 20H
          JMP     RT3       ;RETURN HERE FOR ASCII CODE GREATER THAN 7AH

```

```

;INSTRUCTIONS WHICH FOLLOW ARE EXECUTED AFTER
;VALID PRINTHAMMER FIRING

```

```

;INSTRUCTIONS WHICH FOLLOW ARE EXECUTED FOR
;PRINTWHEEL REPOSITIONING
RT1

```

```

;INSTRUCTIONS WHICH FOLLOW ARE EXECUTED FOR
;ASCII CODE LESS THAN 20H
RT2

```

```

;INSTRUCTIONS WHICH FOLLOW ARE EXECUTED
;FOR ASCII CODE GREATER THAN 7AH
RT3

```


Now for this scheme to work, subroutine PFIR must increment the return address, which is stored in the top two bytes of the stack, every time a conditional Return is executed. Subroutine PFIR is therefore modified as follows:

;PRINTHAMMER FIRING SUBROUTINE

```
PFIR  IN  2  :INPUT I/O PORT 2 TO ACCUMULATOR
      RLC    :MOVE BIT 7 INTO CARRY
      RNC    :IF CARRY IS NOT SET, RETURN
      CALL   INCR  :INCREMENT RETURN ADDRESS
      ANI    20H   :ISOLATE BIT 4 WHICH IS NOW BIT 5
      RZ     :IF ZERO, RETURN
      CALL   INCR  :INCREMENT RETURN ADDRESS
```

;FIRE PRINTHAMMER

```
MVI  A,4  :SET HAMMER PULSE LOW. OUTPUT 0
OUT  3     :TO BIT 2 OF I/O PORT 2
IN   1     :INPUT ASCII CHARACTER CODE TO ACCUMULATOR
ANI  7FH   :MASK OUT HIGH ORDER BIT
```

;COMPARE ASCII CODE WITH LOWEST LEGAL VALUE

```
CPI  20H
RM   :IF CODE IS 1FH OR LESS, BYPASS HAMMER FIRING
CALL INCR  :INCREMENT RETURN ADDRESS
```

;COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE

```
CPI  7AH
RP   :IF CODE IS 7BH OR GREATER, BYPASS FIRING
CALL INCR  :INCREMENT RETURN ADDRESS
```

;ASCII CODE IS VALID

```
MVI  H,03H :LOAD INDEX TABLE ADDRESS, HIGH ORDER BYTE
MOV  L,A    :MOVE LOW ORDER BYTE OF ADDRESS TO L
MOV  L,M    :LOAD LOW ORDER BYTE OF DELAY TABLE ADDRESS
CALL LDLY   :CALL LONG DELAY SUBROUTINE
MVI  A,5    :AT END OF DELAY OUTPUT 1 TO BIT 2 OF
OUT  3      :I/O PORT 2 (HAMMER PULSE HIGH)
```

;EXECUTE A 3 MS PRINTWHEEL RELEASE TIME DELAY

```
LXI  H,MS3
CALL LDLY
```

;OUTPUT 1 TO BIT 0 OF I/O PORT 2. THIS SETS

;PW REL HIGH

```
MVI  A,1
OUT  3
RET   :RETURN FROM SUBROUTINE
```

;LONG DELAY SUBROUTINE. ASSUME H AND L

;ADDRESS THE FIRST OF TWO DATA BYTES WHICH HOLD

;THE INITIAL DELAY CONSTANT

```
LDLY  MOV  E,M :LOAD INITIAL DELAY CONSTANT
      INX  H
```

```
      MOV  D,M
```

;LDLP DCX D :EXECUTE LONG DELAY

```
      MOV  A,D
      ORA  E
      JNZ  LDLP
      RET  :RETURN AT END OF LONG DELAY
```

;SUBROUTINE TO INCREMENT THE RETURN ADDRESS

;OF THE CALLING SUBROUTINE

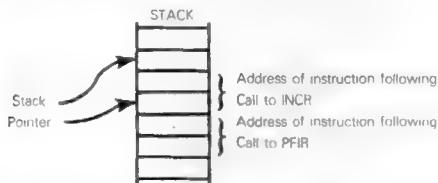
```
INCR  INX  SP :INCREMENT STACK POINTER TWICE
      INX  SP :TO ACCESS PFIR RETURN ADDRESS
      XTHL :EXCHANGE HL WITH PFIR RETURN ADDRESS
```

INX	H	:ADD 3 TO RETURN ADDRESS
INX	H	
INX	H	
XTHL		:RESTORE RETURN ADDRESS
DCX	SP	:DECREMENT STACK POINTER TWICE
DCX	SP	
RET		:RETURN

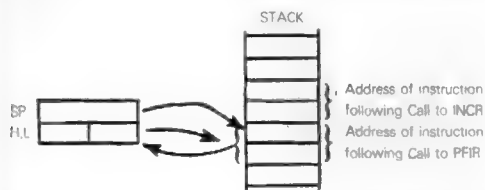
Subroutine INCR is interesting; it shows how the stack may be manipulated. Let us take a look at what happens.

STACK MANIPULATION

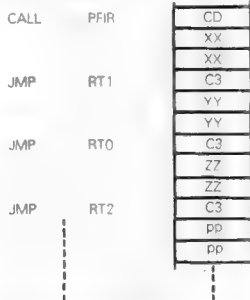
As soon as subroutine INCR is entered, the Stack Pointer contents is increased by two. This has the effect of addressing the PFIR return address rather than the INCR return address:



The XTHL instruction simply saves the contents of the H and L registers at what is now the top of the stack, while moving what was at the top of the stack to the H and L registers:



The next three instructions add 3 to the contents of the H and L registers which now hold the PFIR return address. We add 3 to the return address because, if you look at the calling sequence, a series of Jump instructions follow. Each Jump instruction occupies 3 bytes, which means that each time we bypass a Conditional Return, we must increment the return address by 3:



The next XTHL instruction simply restores the incremented PFIR return address to the top of the stack.

Finally we must restore the Stack Pointer to its original contents, so that the INCR Return instruction will fetch the correct return address.

CONDITIONAL SUBROUTINE CALLS

We are now going to create another subroutine which fires the printhammer, but makes no tests to ensure that the printhammer should be fired. This subroutine simply assumes that a valid ASCII character is in the Accumulator and that the printhammer must be fired. All logic to determine whether printhammer firing is valid is external to the printhammer firing subroutine; therefore this subroutine is called conditionally — so long as all printhammer firing conditions have been met. This is how our program now looks:

```

:TEST PRINTHAMMER FIRING CONDITIONS
LOP3  IN      2      INPUT I/O PORT 2 TO ACCUMULATOR
      RLC          :MOVE BIT 7 INTO CARRY
      JNC      PRD   :IF CARRY IS 0, BYPASS PRINTHAMMER FIRING
      ANI      20H   :ISOLATE BIT 4 WHICH IS NOW BIT 5
      JZ       LOP3  :WAIT FOR NONZERO VALUE BEFORE FIRING

:INPUT CHARACTER TO BE PRINTED
      IN      1      :INPUT ASCII CHARACTER TO ACCUMULATOR
      ANI      7FH   :MASK OUT HIGH ORDER BIT

:COMPARE ASCII CODE WITH LOWEST LEGAL VALUE
      CPI      20H
      JM      PRD    :IF CODE IS 1F OR LESS, BYPASS HAMMER FIRING

:COMPARE ASCII CODE WITH HIGHEST LEGAL VALUE
      CPI      7BH   :IF CODE IS LESS THAN 7BH, CALL
      CM      FIRE   :PRINTHAMMER FIRING SUBROUTINE

:EXECUTE A 2 MILLISECOND PRINTWHEEL READY DELAY
PRD   MVI      A,0   :LOAD ACCUMULATOR WITH 0
```

Notice that the Conditional Return instruction reflects OR programming logic, whereas the Conditional Call instruction reflects AND logic. Thus, subroutine PFIR includes a number of Conditional Return instructions, each of which will execute providing any one invalid condition is encountered. Subroutine FIRE, on the other hand, is called conditionally only when the last of the necessary valid conditions has been tested.

Subroutine FIRE is not shown in detail, since writing it out would add little to the understanding of the Conditional Call instruction. With reference to Figure 4-6, subroutine FIRE would consist of instructions to:

- Set the hammer pulse signal low
- Execute the hammer firing pulse delay
- Set the printhammer firing pulse high
- Execute the 3 millisecond printwheel release time delay
- Output PW REL high

MACROS

When talking about subroutines, we glossed over one consideration — you, the programmer. Subroutines have an additional value in that if they can reduce the number of source program instructions then they will also reduce the amount of time you spend writing the source program since program writing time will be directly proportional to program length.

Let us take another look at the 2 millisecond time delay subroutine. Although in subroutine form the program required more object code bytes, it did not require more instructions:

Old Program			New Program		
LOP2	MVI	A,0		CALL	D2MS
	DCR	A			
	JNZ	LOP2		CALL	D2MS
PDR	MVI	A,0	D2MS	MVI	A,0
LOP6	DCR	A	LOPD	DCR	A
	JNZ	LOP6		JNZ	LOPD
				RET	
6 instructions			6 instructions		
(12 bytes)			(13 bytes, excluding stack and initialization instructions)		

Subroutines can decrease the length of your source program, while increasing the length of your object program, and the program's execution time.

Macros decrease the length of your source program, but have absolutely no effect on your object program.

WHAT IS A MACRO?

A Macro is a form of programming "short hand"; it allows you to define an instruction sequence with a single mnemonic.

Consider the 2 millisecond time delay instruction sequence; we can define it as a macro, labeled D2MS, as follows:

```

D2MS  MACRO
      MVI  A,0
      LOPD DCR  A
      JNZ  LOPD
      ENDM

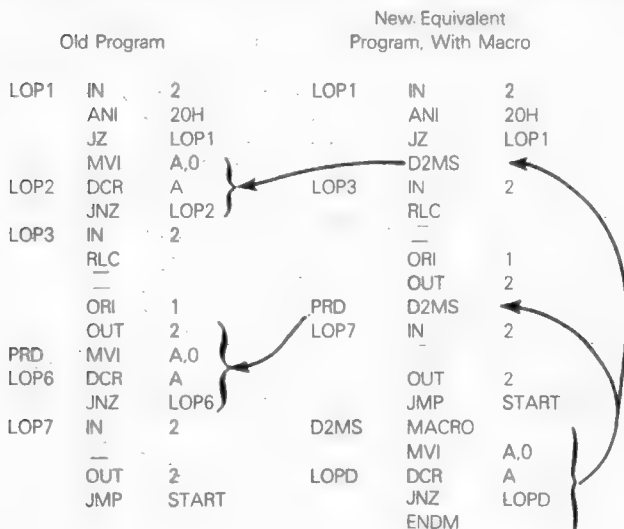
```

**MACRO
DEFINITION**

The two shaded instructions above are, in reality, assembler directives; they bracket a sequence of instructions which henceforth can be identified, as a group, using the label of the MACRO assembler directive.

**MACRO
ASSEMBLER
DIRECTIVES**

This is how we would use the 2 millisecond time delay in our print cycle program:



When the Assembler encounters the symbol D2MS in the mnemonic field, what it does is replace this symbol with the instructions bracketed by directives MACRO and ENDM. The Assembler knows which macro to use in the event that your program has more than one macro, since the symbol in the mnemonic field must be identical to the label of a MACRO directive.

Notice that the Assembler can also do a certain amount of housekeeping associated with the use of macros. The "Old Program" illustrated above has labels LOP2 and LOP6 for the two DCR instructions. The "New Program" has a single label, LOPD, within the macro. The Assembler is smart enough to know that a label appearing within a macro definition must become a series of separate labels when the macro subsequently is inserted a number of times into the source program.

To summarize, you simply take a sequence of repeated instructions, bracket them with MACRO and ENDM directives, then give the macro directive a unique label. Now use the MACRO's label as though it were an instruction mnemonic. The macro definition must appear once and only once, somewhere in the source program.

**MACRO
DEFINITION
LOCATION
IN A SOURCE
PROGRAM**

It is a good idea to collect all of your macros and insert them at the beginning, or at the end of the entire source program.

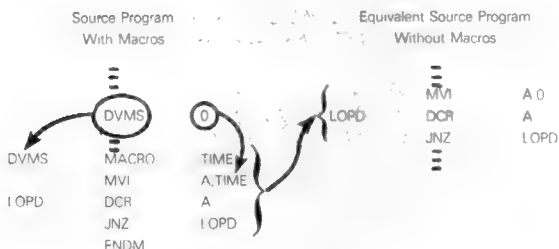
MACROS WITH PARAMETERS

Instructions within a macro can have variable operands; for example, we can create a variable time delay macro as follows:

```
DVMS  MACRO  TIME
      MVI    A,TIME
LOPD  DCR    A
      JNZ    LOPD
      ENDM
```

Symbols appearing in the MACRO directive's operand field are assumed by the Assembler to be "dummy" symbols; the macro reference in the body of the source program must include an equivalent operand field. The Assembler will equate the macro reference's operand field to the MACRO directive's operand field, and make substitutions accordingly.

This is how the substitution works:



Here is another example; the macro reference:

```

    DVMS    80H

```

is equivalent to:

```

    MVI    A, 80H
LOPD    DCR    A
        JNZ    LOPD

```

Depending on whose Assembler you are using, you can play interesting games with the macro parameter list, in theory (but not always in practice), there are no restrictions on the length or nature of the macro parameter list. Suppose you want to vary the register used in the time delay instruction sequence; some assemblers will let you do so as follows:

```

    DVMS    C, 3CH
    MACRO X, TIME
    MVI     X, TIME
LOPD    DCR    X
        JNZ    LOPD
    ENDM

```

The Assembler will substitute:

```

    DVMS    C, 3CH

```

with:

```

    MVI     C, 3CH
LOPD    DCR    C
        JNZ    LOPD

```

You will have to read the Assembler manual that accompanies your development system in order to know the exact macro features available to you.

INTERRUPTS

It would be hard to justify including interrupts within the microcomputer system developed in Chapter 4. In fact, interrupts should be used quite sparingly in microcomputer applications.

We will not enter into a long discussion on the strengths and weaknesses of interrupts within microcomputer systems, that subject has been adequately covered in "An Introduction To Microcomputers", Volume I. To summarize, however, recall that interrupts are a valid tool within microcomputer systems only when dealing with fast, asynchronous events

**WHEN TO USE
INTERRUPTS**

Now having issued a warning against the indiscriminate use of interrupts, we will proceed to incorporate simple interrupt processing into our microcomputer program in the interests of demonstrating how it is done.

INTERRUPT HARDWARE CONSIDERATIONS

For an interrupt to be processed within an 8080 microcomputer system, an interrupt request signal must be input high to the CPU at a time when interrupts have been enabled.

Interrupts are enabled and disabled by executing EI and DI instructions, respectively. The enabled condition is identified by a high output from the 8080 CPU INT ENABLE signal (INTE). External logic does not have to interrogate this signal before attempting to request an interrupt, any interrupt request will simply be ignored by the CPU while interrupts have been disabled

**INTERRUPT
ENABLE**

If an interrupt request is received while interrupts have been enabled, then upon completing execution of the current instruction, the CPU will output an interrupt acknowledge signal via the 8228 System Controller. External logic which is requesting the interrupt must respond to the interrupt acknowledge by inputting an 8-bit instruction

**INTERRUPT
ACKNOWLEDGE
RESTART
INSTRUCTION**

code which is going to be interpreted as the instruction code to be executed next. Usually one of the eight possible Restart instruction codes will be fetched. These instructions are equivalent to single byte subroutine calls; they cause the contents of the Program Counter to be pushed onto the stack, after which program execution continues at a low memory address which may be computed as follows

RST N instruction code: 1 1 1 XXX 1 1

0 0 0	N = 0
0 0 1	N = 1
0 1 0	N = 2
0 1 1	N = 3
1 0 0	N = 4
1 0 1	N = 5
1 1 0	N = 6
1 1 1	N = 7

New Program

Counter Contents 0 0 0 0 0 0 0 0 0 0 XXX 0 0 0

Therefore RST N instructions are equivalent to subroutine CALL instructions, with program execution branching as follows:

Subroutine
 RST 0 branch to 0000₁₆
 RST 1 branch to 0008₁₆
 RST 2 branch to 0010₁₆
 RST 3 branch to 0018₁₆
 RST 4 branch to 0020₁₆
 RST 5 branch to 0028₁₆
 RST 6 branch to 0030₁₆
 RST 7 branch to 0038₁₆

The simplest way of creating an appropriate RST instruction externally is via an 8212 I/O port. Necessary logic is illustrated in Figure 5-1.

RST INSTRUCTION CODE CREATION

Now in Figure 5-1 the 8212 I/O port has been used in the most elementary way possible. Let us examine this simple I/O port use.

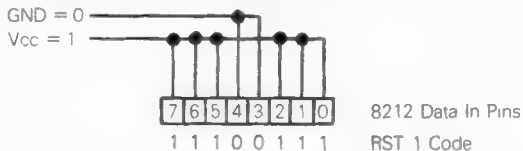
Since there is but one interrupt, it is tied directly to the INT REQ 8080 CPU input. The INT ENABLE CPU output is simply ignored.

SINGLE INTERRUPT CONFIGURATION

When the single external logic source which can request an interrupt does so by inputting INT REQ high, at some subsequent time the CPU will acknowledge the interrupt via the 8228 Bus Controller, by outputting $\overline{\text{INTA}}$ low on the Control Bus. Now at the 8212 I/O port the $\overline{\text{INTA}}$ signal is used as one of the two device select signals. The other device select signal, DS2, must be high; it is therefore tied to Vcc.

The sole purpose of the 8212 I/O port in Figure 5-1 is to input an RST 1 instruction once an interrupt is acknowledged by $\overline{\text{INTA}}$ being output high. We select the RST 1 instruction arbitrarily. **The only RST instruction we cannot select is RST 0, since memory location 0 is used following a reset.** The RST 1 instruction code is generated by tying appropriate data-in pins to ground and to level 1 current:

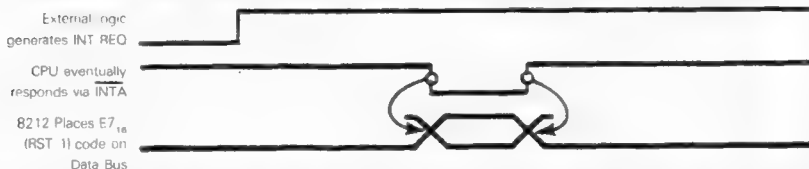
8212 I/O PORT USED IN INTERRUPT SYSTEM



The mode pin (MD) is tied to ground, since the 8212 I/O port is being used in input mode.

Since the only criterion for the 8212 I/O port to output data is that $\overline{\text{INTA}}$ be low, the STROBE and CLEAR inputs are disabled by tying them to Vcc.

In summary, this is what happens when external logic requests an interrupt:



You, as a logic designer or programmer, do not need to concern yourself with Data Bus timing. The $\overline{\text{INTA}}$ signal will correctly strobe the RST 1 instruction code into the CPU, causing the 8212 I/O port output to be interpreted as an instruction code, rather than being interpreted as data.

Even a brief examination of the way in which the 8212 I/O port has been incorporated into our microcomputer system will make it obvious that there are plenty of more elaborate ways in which the 8212 I/O port could be used; for example, it can handle multiple interrupts.

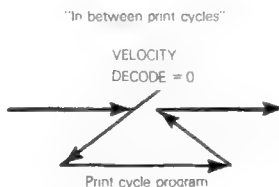
Tying the external interrupt request directly to the 8080 CPU is a primitive method of handling interrupts, but quite adequate in a microcomputer system where only one external device is likely to request an interrupt.

Before we look at some of the more elaborate interrupt request schemes, let us look at the programming considerations associated with the simple scheme we have illustrated in Figure 5-1.

INTERRUPT SERVICE PROGRAM

First of all, **how are we going to use the interrupt?**

We could assume that the microcomputer system is being used to do more than implement print cycle logic. **Suppose there is a great deal of routine housekeeping logic required by the printer interface, with the result that the entire print cycle can be looked upon as an intermittent asynchronous event. Now, instead of having our program execute an "in between print cycles" instruction loop, we will assume that some other program is being continuously executed in between print cycles. Execution of the print cycle program is triggered by the VELOCITY DECODE signal, the inverse of which is tied to INT REQ in Figure 5-1. This is the instruction execution pattern which results:**



Here is the instruction sequence required to execute the print cycle program which follows an interrupt derived from the inverse of velocity decode:

```

ORG      8
:ORIGIN PRINT CYCLE PROGRAM INTERRUPT SERVICE ROUTINE
:AT 0008H, SINCE THIS IS THE EXECUTION ADDRESS
:WHICH FOLLOWS EXECUTION OF AN RST 1 INSTRUCTION
    CALL   START    ;CALL PRINT CYCLE PROGRAM AS A SUBROUTINE
    RET     ;RETURN FROM INTERRUPT
ORG      NNNN
:SELECT ANY VIABLE ORIGIN FOR THE PRINT CYCLE PROGRAM
:INITIALIZE PRINT CYCLE. OUTPUT 0 TO BITS 0 AND 1 OF
:I/O PORT 2. OUTPUT 1 TO BIT 3 OF I/O PORT 2
START    MVI     A,0CH    ;LOAD MASK INTO ACCUMULATOR
        OUT     2        ;OUTPUT TO I/O PORT 2
        -
        -
        -
:AT END OF PRINT CYCLE, SET BIT 1 OF I/O PORT 2 TO 1
:THIS SETS CH RDY HIGH
        MVI     A,3
        OUT     3
        RET

```

Notice that the "in between print cycles" instructions have been removed; START now identifies the first instruction of the print cycle itself.

The Origin specified for the print cycle program is unimportant. We do not know what other programs are being executed within the microcomputer system, or where these other programs may reside in program memory; therefore we cannot assign memory space to the print cycle program at this time. When you actually implement the entire microcomputer system, you must carefully map out exactly where in memory every program resides, but for the purposes of our current illustration, this is a completely unimportant consideration.

INTERRUPT PROGRAM ORIGIN

Notice that the final instructions of the print cycle use the bit set control instructions to modify CH RDY; but strobed I/O, using the 8255 I/O Port 1 in Mode 1 has not been assumed.

The final JMP START instruction is replaced by a simple RETURN instruction, since the entire print cycle program was, in effect, called as a subroutine.

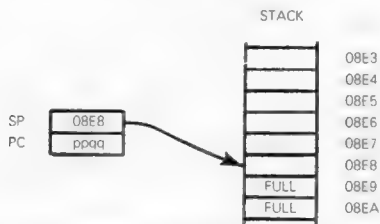
The microcomputer system tracks itself around memory following an interrupt using the stack. Let us explain how.

STACK TOP

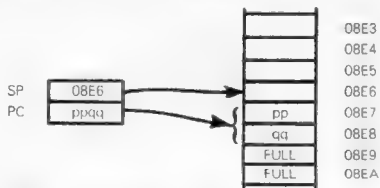
When we last looked at the stack, it was originated at 08FF₁₆; this was the top of the read/write memory and the stack had not been accessed. We will now assume that the stack has been accessed by whatever program is executed in between print cycles, so that when the print cycle gets executed, the Stack Pointer contains the address 08E8₁₆. We are simply assuming that there has been some level of stack activity, but what we neither know nor care.

Following the interrupt acknowledge, this is how the stack is used:

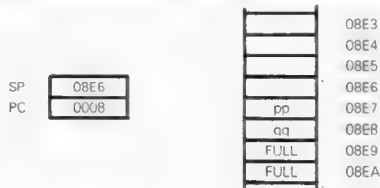
- 1) When the VELOCITY DECODE signal requests an interrupt, this is the situation:



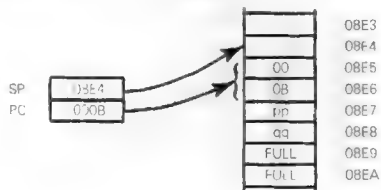
- 2) The interrupt is acknowledged. First the Program Counter contents are saved on the stack:



- 3) Next the RST 1 instruction causes 0008 to be loaded into the Program Counter:

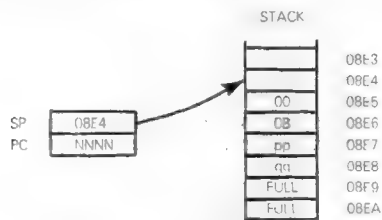


- 4) At memory location 0008 there is a CALL START instruction. This causes the address of the next instruction to be saved on the stack:

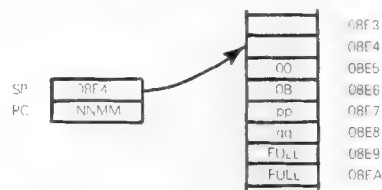


Remember, the next (RET) instruction is located at 000B₁₆ because the CALL START instruction occupies three bytes, 0008₁₆, 0009₁₆ and 000A₁₆.

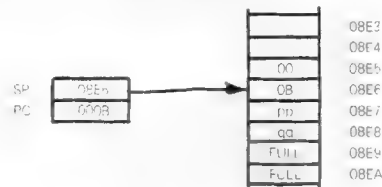
- 5) Now the address of the instruction labeled START is loaded into the Program Counter. This is the first instruction of the print cycle routine:



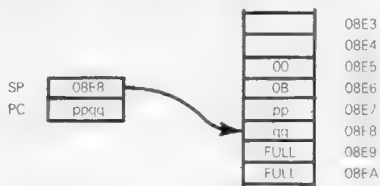
- 6) Assume the final RET instruction in the print cycle routine is stored in memory location NNNM. When this RET instruction is about to be executed, this is the situation:



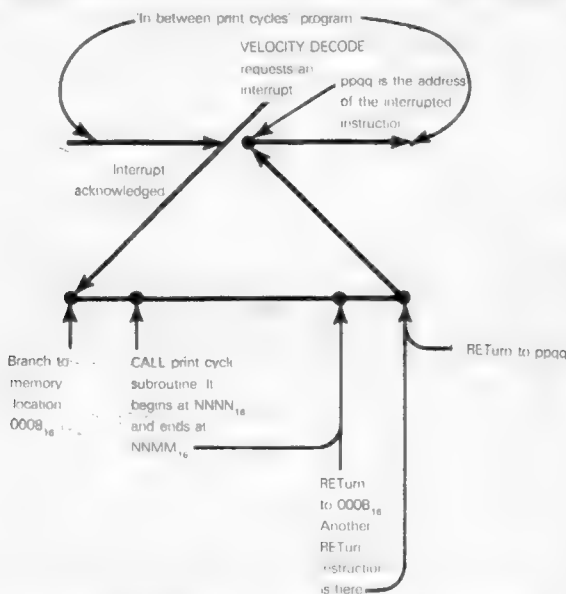
- 7) When this RET instruction executes, the Stack Pointer increments by 2, and the top two stack bytes are moved to the Program Counter:



- 8) Another RET instruction is stored at memory location 000B, so the same thing happens again:



The address of the interrupted instruction has been restored. Here is a complete illustration



The method we have just described for processing an interrupt has been kept simple to make sure you have no trouble following the program execution paths; but the program will not work. We have shown a background program being interrupted in order to execute the print cycle routine; but when does the background program get interrupted? Remember, the program which is interrupted is sharing the same CPU and the same registers with the print cycle program. We have to assume that the interrupted program has useful information stored in the registers and perhaps the status flags have meaning which must be preserved. Given the interrupt service program illustrated thus far, when we return from the print cycle program to the interrupted program, we are giving the interrupted program whatever arbitrary register contents the print cycle program finishes up with. That will never do. **We must therefore bracket the print cycle execution program with instructions that save the contents of registers and status on the stack — before modifying a**

SAVING REGISTERS AND STATUS

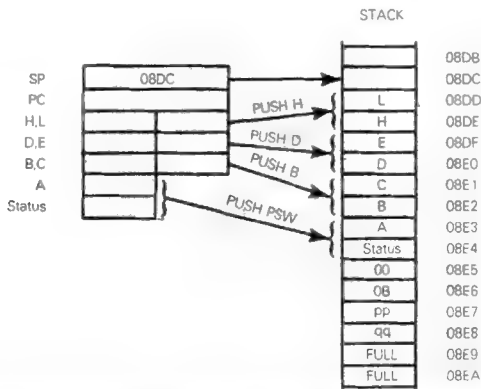
single register or status; at the end of the program, original registers and status contents must be restored. This is how the print cycle program now looks:

```

ORG      8
;ORIGIN PRINT CYCLE PROGRAM INTERRUPT SERVICE ROUTINE
;AT 0008H, SINCE THIS IS THE EXECUTION ADDRESS
;WHICH FOLLOWS EXECUTION OF AN RST 1 INSTRUCTION
CALL     START    ;CALL PRINT CYCLE PROGRAM AS A SUBROUTINE
RET      ;RETURN FROM INTERRUPT
ORG      NNNN
;PUSH CONTENTS OF ALL REGISTERS AND STATUS ONTO STACK
START    PUSH     PSW      ;SAVE ACCUMULATOR AND STATUS
        PUSH     B        ;SAVE B AND C
        PUSH     D        ;SAVE D AND E
        PUSH     H        ;SAVE H AND L
;SELECT ANY VIABLE ORIGIN FOR THE PRINT CYCLE PROGRAM
;INITIALIZE PRINT CYCLE: OUTPUT 0 TO BITS 0 AND 1 OF
;I/O PORT 2: OUTPUT 1 TO BIT 3, OF I/O PORT 2
        MVI      A,0CH    ;LOAD MASK INTO ACCUMULATOR
        OUT      2        ;OUTPUT TO I/O PORT 2
        --
;AT END OF PRINT CYCLE: SET BIT 1 OF I/O PORT 2 TO 1
;THIS SETS CH RDY HIGH
        MVI      A,3
        OUT      3
;RESTORE INTERRUPTED PROGRAM'S REGISTERS CONTENTS AND STATUS
        POP      H        ;RESTORE H AND L
        POP      D        ;RESTORE D AND E
        POP      B        ;RESTORE B AND C
        POP      PSW      ;RESTORE ACCUMULATOR AND STATUS
        RET

```

The way in which data gets saved on the stack is straightforward enough:



So long as you remember to pop registers and status contents in the reverse order from which you pushed them, you will have no problems.

JUSTIFYING INTERRUPTS

Minicomputer programmers and large computer programmers make indiscriminate use of interrupts simply to share the cost of the Central Processing Unit among a number of different applications.

You, as a microcomputer user, are going to have to justify sharing a cost which may range between \$5 and \$20. Against this cost you must charge the cost of external logic needed to create interrupt request signals and restart instructions — as well as the extra cost of programming. **The economic tradeoff makes it far from obvious that interrupts are viable within microcomputer systems.** You must examine your application with care before assuming out of hand that interrupts represent the way to go. A second CPU, or an entire second microcomputer system will frequently be cheaper than using interrupts to share a single microcomputer system between a number of different applications.

INTERRUPT ECONOMICS

Assuming that interrupts look economical for your application, timing considerations are also important.

INTERRUPT TIMING CONSIDERATIONS

Certainly interrupts look very attractive when your application is handling asynchronous events. In our case, **suppose the average print cycle lasts approximately 10 milliseconds; also, suppose it is impossible to say whether the time interval between print cycles will be 1 millisecond or 100 milliseconds. Under these circumstances, in order to execute some other program in the time in between print cycles, we must use interrupts to initiate the print cycle** — since we have no idea when the next print cycle is to begin.

In reality, the time which elapses between print cycles will be very accurately known. **A printer will have some advertised character printing rate.** If this rate is 45 characters per second, then 22.2 milliseconds will be required per printed character. If 10 of the 22 milliseconds are needed to execute the actual print cycle routine, then 12 milliseconds will remain in between print cycles. **We no longer need interrupts.** So long as the program which executes in between print cycles is broken into segments, each of which executes in 12 milliseconds or less, then each segment can terminate with an instruction loop which tests the status of the velocity decode input in order to initiate the next print cycle:

```
LOOP      IN      2      :INPUT I/O PORT 2 CONTENTS
          ANI      20H    :ISOLATE VELOCITY DECODE SIGNAL
          JNZ      LOOP   :IF STILL 1, NEW PRINT CYCLE HAS NOT BEGUN
```

There is a time penalty associated with every interrupt that gets processed.

INTERRUPT SERVICE TIME LOSS

Look at the instructions which must be executed before and after the print cycle program itself: there are 4 Push instructions, 4 Pop instructions, a Call instruction and a Return instruction. Add up the number of cycles required and you will find that 111 cycles are needed to execute all of these instructions — and that means **55.5 microseconds per interrupt.**

That is approximately 5% of the total time required to execute the entire print cycle program.

Project this "overhead" time into a more complex system, where perhaps 10 different external logic sources may generate an interrupt request. Such a complex interrupt scheme may sound reasonable to a minicomputer programmer, but within the microcomputer system it is possible for 555 microseconds to be consumed by unproductive "overhead" time. 50% of the time your microcomputer system may be doing nothing more than saving and restoring registers and status. Clearly, you must approach interrupts with an element of caution.

MULTIPLE INTERRUPTS

Suppose your application is one in which all the warnings against use of multiple interrupts do not apply. You may, for example, have an application in which a number of asynchronous events occur with sufficient infrequency that they do not seriously impact on available execution time. **There are innumerable ways in which multiple interrupts can be implemented in an 8080-type microcomputer system** and it is certainly beyond the scope of this book to explore them all. There are, however, two basic concepts to all multiple interrupt schemes.

- 1) Instead of tying external interrupt requests directly to the interrupt request pin of the CPU, the 8255 PPI or the 8212 I/O port is used to buffer and transmit interrupt requests.
- 2) Providing eight or fewer external interrupts are being processed, the Restart instructions adequately discriminate between interrupts. An 8212 I/O port can be used with a one-of-eight decoder to generate the appropriate Restart instruction as follows:

If you have eight external interrupts, remember that one of them is going to use the RST 0 Restart instruction — and this will coincide with the Reset condition, since both cause program execution to branch to memory location 0.

If you have seven or fewer different external interrupts, processing them is relatively straightforward.

If you have nine or more external interrupts and you are trying to process them using one CPU, in all probability there is something wrong with the way in which you are designing your microcomputer system. It is possible that someone will think up an application where such a complex interrupt scheme is economically implemented via a microcomputer system with one CPU. If yours is such a system, we would like to hear about it.

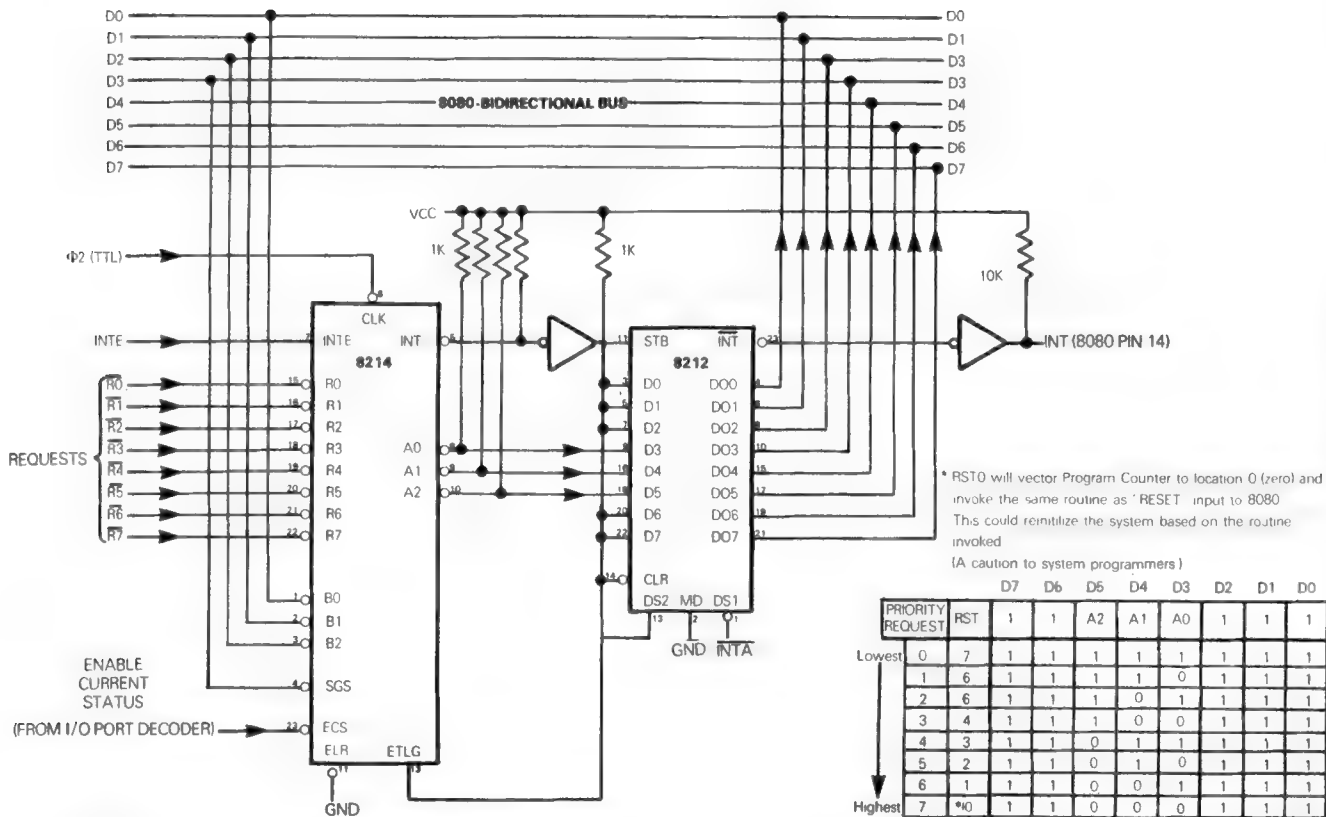


Figure 5-2. Generation Of The Restart Instruction Code Following An Interrupt. For An 8080 Microcomputer System

Chapter 6

THE 8080/9080 INSTRUCTION SET

Instructions falsely frighten microcomputer users who are new to programming. Taken as an isolated event, operations associated with the execution of a single instruction are easy enough to follow — and that is the purpose of this chapter.

Why are the instructions of a microcomputer referred to as an instruction “set”? Because the instructions selected by the designers of any microcomputer are selected with great care; it must be easy to execute complex operations as a sequence of simple events — each of which is represented by one instruction from a well designed instruction “set”.

Remaining consistent with “An Introduction To Microcomputers, Volume II”, Table 6-1 summarizes the 8080/9080 microcomputer instruction set, with similar instructions grouped together.

Individual instructions are described next in alphabetic order of instruction mnemonic.

In addition to simply stating what each instruction does, the purpose of the instruction within normal programming logic is identified.

ABBREVIATIONS

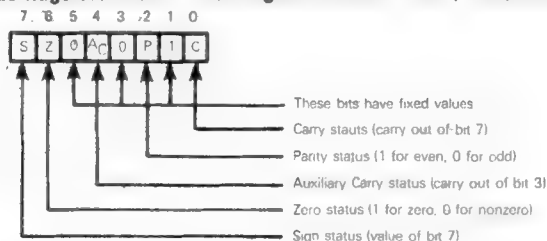
These are the abbreviations used in this chapter:

A	The Accumulator	
B	The B register	} These are sometimes referred to as a register pair
C	The C register	
D	The D register	
E	The E register	} These are sometimes referred to as a register pair
H	The H register	
L	The L register	} This register pair provides the implied memory address
CS	Carry status	
AC	Auxiliary carry status	
ZS	Zero status	
SS	Sign status	
PS	Parity status	
I	The Instruction register	
I2	Second object code byte	
I3	Third object code byte	
PC	The Program Counter	
SP	The Stack Pointer	
PSW	The Program Status Word, which has bits assigned to status flags as shown on the next page.	
H	Appearing at the end of a group of digits (e.g., 213AH) specifies hexadecimal digits.	
DATA	8-bit immediate data	
DEV	An I/O device	
DATA16	16-bit immediate data	
REG	Register A, B, C, D, E, H or L	
M	Memory, address implied by HL	
LABEL	A 16-bit address, specifying an instruction label	
RP	A register pair: B for BC, D for DE, H for HL, SP for Stack Pointer	

PORT : An I/O port, identified by a number between 0 and FF₁₆.
ADDR : A 16-bit address, specifying a data memory byte.
[] : Contents of location identified within brackets
[0] : Memory byte addressed by location identified within brackets
→ : Move data in direction of arrow
↔ : Exchange contents of locations on either side of arrow
+ : Add
- : Subtract
Λ : AND
∨ : OR
⊕ : XOR

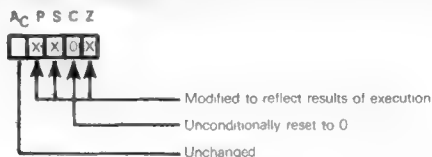
STATUS

The five status flags are stored in a Program Status Word (PSW) as follows:



PSW and A are sometimes treated as a register pair

The effect of instruction execution on status is illustrated as follows:



Within instruction execution illustrations, an X identifies a status that is set or reset. A O identifies a status that is always cleared. A blank means the status does not change.

**STATUS
CHANGES
WITH
INSTRUCTION
EXECUTION**

INSTRUCTION OBJECT CODES

Instruction object codes are represented as 2 hexadecimal digits for instructions without variations.

Instruction object codes are represented as 8 binary digits for instructions with variations; the binary digit representation of variations is then identifiable.

INSTRUCTION EXECUTION TIMES AND CODES

Table 6-2 lists instructions in alphabetic order, showing object codes and execution times, expressed as machine cycles.

Where two instruction cycles are shown, the first is for "condition not met" whereas the second is for "condition met".

Table 6-1. A Summary Of 8080/9080 Microcomputer Instruction Set

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES					OPERATION PERFORMED
				C	AC	Z	S	P	
I/O	IN	DEV	2						[A] ← [DEV] Input to A from device DEV (DEV = 0 to 255).
	OUT	DEV	2						[DEV] ← [A] Output from A to device DEV (DEV = 0 to 255).
Primary Memory Reference	LDAX	RP	1						[A] ← [[RP]] Load A using address implied by BC (RP = B) or DE (RP = D).
	STAX	RP	1						[[RP]] ← [A] Store A using implied addressing as for LDAX.
	MOV	R,M	1						[R] ← [[H,L]] Load any register using address implied by HL.
	MOV	M,R	1						[[H,L]] ← [R] Store any register using address implied by HL.
	LDA	ADDR	3						[A] ← [ADDR], i.e., [A] ← [[I3,I2]] Load A, use direct addressing.
	STA	ADDR	3						[ADDR] ← [A], i.e., [[I3,I2]] ← [A] Store A, use direct addressing.
	LHLD	ADDR	3						[L] ← [ADDR], [H] ← [ADDR + 1] i.e., [L] ← [[I3,I2]], [H] ← [[I3,I2] + 1] Load H and L registers, use direct addressing.
	SHLD	ADDR	3						[ADDR] ← [L], [ADDR + 1] ← [H] i.e., [[I3,I2]] ← [L], [[I3,I2] + 1] ← [H] Store H and L registers, use direct addressing.

Table 6-1. A Summary of 8080/9080 Microcomputer Instruction Set
(Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES					OPERATION PERFORMED
				C	AC	Z	S	P	
Secondary Mem Ref (Memory Operate)	ADD	M	1	X	X	X	X	X	$[A] \leftarrow [A] + [[H,L]]$ Add to A
	ADC	M	1	X	X	X	X	X	$[A] \leftarrow [A] + [[H,L]] + [CS]$ Add with Carry to A
	SUB	M	1	X	X	X	X	X	$[A] \leftarrow [A] - [[H,L]]$ Subtract from A
	SBB	M	1	X	X	X	X	X	$[A] \leftarrow [A] - [[H,L]] - [CS]$ Subtract from A with borrow
	ANA	M	1	0	X	X	X	X	$[A] \leftarrow [A] \wedge [H,L]$ AND with A
	XRA	M	1	0	X	X	X	X	$[A] \leftarrow [A] \vee [H,L]$ Exclusive-OR with A
	ORA	M	1	0	X	X	X	X	$[A] \leftarrow [A] \vee [H,L]$ OR with A
	CMP	M	1	X	X	X	X	X	Compare with A
	INR	M	1		X	X	X	X	$[[H,L]] \leftarrow [[H,L]] + 1$ Increment memory
Immediate	DCR	M	1		X	X	X	X	$[[H,L]] \leftarrow [[H,L]] - 1$ Decrement memory
	LXI	RP, DATA16	3						$[RP] \leftarrow \text{DATA16}$ Load 16-bit immediate data into BC (RP = B), DE (RP = D), HL (RP = H) or SP (RP = SP)
	MVI	M, DATA	2						$[[H,L]] \leftarrow \text{DATA}$ Load 8-bit immediate data into memory location with address implied by HL.
	MVI	R, DATA	2						$[R] \leftarrow \text{DATA}$ Load 8-bit immediate data into any register

Table 6-1. A Summary Of 8080/9080 Microcomputer Instruction Set
(Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES					OPERATION PERFORMED
				C	AC	Z	S	P	
Jump	JMP	ADDR	3						[PC] ← ADDR Jump to instruction with label ADDR.
	PCHL		1						[PC] ← [H,L] Jump to instruction at location implied by HL.
Subroutine Call and Return (Immediate and Stack)	CALL	ADDR	3						[[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2 Jump to subroutine starting at ADDR
	CC	ADDR	3						[[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2 Jump to subroutine if C = 1
	CNC	ADDR	3						[[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2 Jump to subroutine if C = 0
	CZ	ADDR	3						[[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2 Jump to subroutine if Z = 1
	CNZ	ADDR	3						[[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2 Jump to subroutine if Z = 0
	CP	ADDR	3						[[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2 Jump to subroutine if S = 0
	CM	ADDR	3						[[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2 Jump to subroutine if S = 1
	CPE	ADDR	3						[[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2 Jump to subroutine if even parity
	CPO	ADDR	3						[[SP]] ← [PC], [PC] ← ADDR, [SP] ← [SP] - 2 Jump to subroutine if odd parity
	RET		1						[PC] ← [[SP]], [SP] ← [SP] + 2 Return from subroutine

Table 6-1. A Summary Of 8080/9080 Microcomputer Instruction Set
(Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES					OPERATION PERFORMED
				C	AC	Z	S	P	
Subroutine Call and Return (Immediate and Stack)	RC		1						$[PC] \leftarrow [[SP]], [SP] \leftarrow [SP] + 2$ Return from subroutine if C = 1
	RNC		1						$[PC] \leftarrow [[SP]], [SP] \leftarrow [SP] + 2$ Return from subroutine if C = 0
	RZ		1						$[PC] \leftarrow [[SP]], [SP] \leftarrow [SP] + 2$ Return from subroutine if Z = 1
	RNZ		1						$[PC] \leftarrow [[SP]], [SP] \leftarrow [SP] + 2$ Return from subroutine if Z = 0
	RM		1						$[PC] \leftarrow [[SP]], [SP] \leftarrow [SP] + 2$ Return from subroutine if S = 1
	RP		1						$[PC] \leftarrow [[SP]], [SP] \leftarrow [SP] + 2$ Return from subroutine if S = 0
	RPE		1						$[PC] \leftarrow [[SP]], [SP] \leftarrow [SP] + 2$ Return from subroutine if even parity
	RPO		1						$[PC] \leftarrow [[SP]], [SP] \leftarrow [SP] + 2$ Return from subroutine if odd parity
Immediate Operate	ADI	DATA	2	X	X	X	X	X	$[A] \leftarrow [A] + \text{DATA}$ Add immediate to A
	ACI	DATA	2	X	X	X	X	X	$[A] \leftarrow [A] + \text{DATA} + [CS]$ Add with carry immediate to A
	SUI	DATA	2	X	X	X	X	X	$[A] \leftarrow [A] - \text{DATA}$ Subtract immediate from A
	SBI	DATA	2	X	X	X	X	X	$[A] \leftarrow [A] - \text{DATA} - [CS]$ Subtract immediate with borrow from A
	ANI	DATA	2	0	X	X	X	X	$[A] \leftarrow [A] \wedge \text{DATA}$ AND immediate with A

Table 6-1. A Summary Of 8080/9080 Microcomputer Instruction Set
(Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES					OPERATION PERFORMED
				C	AC	Z	S	P	
Immediate Operate	XRI	DATA	2	0	0	X	X	X	[A] ← [A] ∨ DATA Exclusive-OR immediate with A
	ORI	DATA	2	0	0	X	X	X	[A] ← [A] ∨ DATA OR immediate with A
	CPI	DATA	2	X	X	X	X	X	Compare immediate with A
Jump On Condition	JC	ADDR	3						[PC] ← ADDR Jump if C = 1
	JNC	ADDR	3						[PC] ← ADDR Jump if C = 0
	JZ	ADDR	3						[PC] ← ADDR Jump if Z = 1
	JNZ	ADDR	3						[PC] ← ADDR Jump if Z = 0
	JP	ADDR	3						[PC] ← ADDR Jump if S = 0
	JM	ADDR	3						[PC] ← ADDR Jump if S = 1
	JPE	ADDR	3						[PC] ← ADDR Jump on even parity
	JPO	ADDR	3						[PC] ← ADDR Jump on odd parity

Table 6-1. A Summary Of 8080/9080 Microcomputer Instruction Set
(Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES					OPERATION PERFORMED
				C	AC	Z	S	P	
Reg-Reg Move	MOV	D,S	1						[R] ← [R] Move any register (S) to any register (D)
	XCHG		1						[D] ↔ [H], [E] ↔ [L] Exchange DE with HL
	SPHL		1						[H,L] ↔ [SP] Exchange SP with HL
Register-Register Operate	ADD	R	1	X	X	X	X	X	[A] ← [A] + [R] Add any register to A
	ADC	R	1	X	X	X	X	X	[A] ← [A] + [R] + [CS] Add with Carry any register to A
	SUB	R	1	X	X	X	X	X	[A] ← [A] - [R] Subtract any register from A
	SBB	R	1	X	X	X	X	X	[A] ← [A] - [R] - [CS] Subtract any register with borrow from A
	ANA	R	1	0	X	X	X	X	[A] ← [A] ∧ [R] AND any register with A
	XRA	R	1	0	X	X	X	X	[A] ← [A] ⊕ [R] Exclusive-OR any register with A
	ORA	R	1	0	X	X	X	X	[A] ← [A] ∨ [R] OR any register with A
	CMP	R	1	X	X	X	X	X	Compare any register with A
Register Operate	INR	R	1		X	X	X	X	[R] ← [R] + 1 Increment any register
	DCR	R	1		X	X	X	X	[R] ← [R] - 1 Decrement any register
	CMA		1						[A] ← [A]̅ Complement A

Table 6-1. A Summary Of 8080/9080 Microcomputer Instruction Set
(Continued)



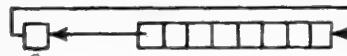

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES						OPERATION PERFORMED
				C	AC	Z	S	P		
Register Operate	DAA		1	X	X	X	X	X	Decimal adjust A	
	RLC		1	X					 Rotate A left with branch carry	
	RRC		1	X					 Rotate A right with branch carry	
	RAL		1	X					 Rotate A left with carry	
	RAR		1	X					 Rotate A right with carry	
	DAD	RP	1	X					$[H.L] \leftarrow [H.L] + [RP]$ Add to HL	
	INX	RP	1						$[RP] \leftarrow [RP] + 1$ Increment RP. RP = BC, DE, HL or SP	
	DCX	RP	1						$[RP] \leftarrow [RP] - 1$ Decrement RP	
Stack	PUSH	RP	1						$\left. \begin{array}{l} [[SP]] \leftarrow [RP], [SP] \leftarrow [SP] - 2 \\ \text{Push RP contents onto stack} \\ [RP] \leftarrow [[SP]], [SP] \leftarrow [SP] + 2 \\ \text{Pop stack into RP} \end{array} \right\} RP = BC, DE, HL \text{ or } PSW$	
	POP	RP	1							
	XTHL		1						$[H.L] \longleftrightarrow [[SP]]$ Exchange HL with top of stack	

Table 6-1. A Summary Of 8080/9080 Microcomputer Instruction Set
(Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES	OPERATION PERFORMED
				C AC Z S P	
Interrupt	EI		1		Enable interrupts
	DI		1		Disable interrupts
	RST		1		Restart
Status	STC		1	1	[CS] ← 1 Set Carry
	CMC		1	X	[CS] ← [CS] Complement Carry
	NOP		1		No operation
	HLT		1		Halt

Statuses: C = Carry

AC = Carry out of bit 3

Z = Zero

S = Sign

P = Parity

X = Status set or reset

0 = Status reset

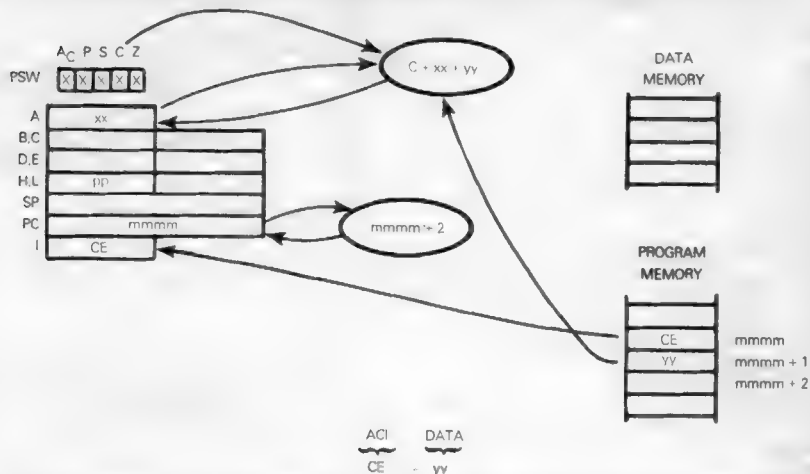
Blank = Status unchanged

Table 6-2. A Summary Of Instruction Object Codes
And Execution Cycles

INSTRUCTION	OBJECT CODE	BYTES	CYCLES	INSTRUCTION	OBJECT CODE	BYTES	CYCLES
ACI DATA	CF YY	2	7	LXI RP DATA 16	00XX0001	3	10
ADC REG	10001XXX	1	4		YYYY		
ADC M	8E	1	7	MOV REG REG	01dddsss	1	5
ADD REG	10000XXX	1	4	MOV M REG	01110sss	1	7
ADD M	8E	1	7	MOV REG, M	01ddd110	1	7
ADI DATA	C6 YY	2	7	MVI REG DATA	00ddd110	2	7
ANA RFG	10100XXX	1	4		vv		
ANA M	A6	1	7	MVI M DATA	36 YY	2	10
ANI DATA	E6 YY	2	7	NOP	00	1	4
CALL LABEL	CD ppqq	3	17	ORA REG	10110XXX	1	5
CC LABEL	DC ppqq	3	11/17	ORA M	B6	1	7
CM LABEL	FC ppqq	3	11/17	ORI DATA	F6 YY	2	7
CMA	2F	1	4	OUT PORT	D3 YY	2	10
CMC	3F	1	4	PCHL	F9	1	5
CMP REG	10111XXX	1	4	POP RP	11XX0001	1	10
CMP M	BE	1	7	PUSH RP	11XX0101	1	11
CNC LABEL	D4 ppqq	3	11/17	RAL	17	1	4
CNZ LABEL	C4 ppqq	3	11/17	RAR	1F	1	4
CP LABEL	F4 ppqq	3	11/17	RC	D8	1	5/11
CPE LABEL	EC ppqq	3	11/17	RET	C9	1	10
CPI DATA	FE YY	2	7	RLC	07	1	4
CPO LABEL	E4 ppqq	3	11/17	RM	F8	1	5/11
CZ LABEL	CC ppqq	3	11/17	RNC	D0	1	5/11
DAA	27	1	4	RNZ	C0	1	5/11
DAD RP	00XX1001	1	10	RP	F0	1	5/11
DCR REG	00XXX101	1	5	RPE	E8	1	5/11
DCR M	35	1	10	RPO	F0	1	5/11
DCX RP	00XX1011	1	5	RRC	0F	1	4
DI	F3	1	4	RST N	11XXX111	1	11
EI	FB	1	4	RZ	C8	1	5/11
HLT	75	1	4	SBB REG	10011XXX	1	4
IN PORT	DB YY	2	10	SBB M	9E	1	7
INR REG	00XXX100	1	5	SBI DATA	DE YY	2	7
INR M	34	1	10	SHLD ADDR	22 ppqq	3	16
INX RP	00XX0011	1	5	SPHL	F9	1	5
JC LABEL	DA ppqq	3	10	STA ADDR	32 ppqq	3	13
JM LABEL	FA ppqq	3	10	STAX RP	000X0010	1	7
JMP LABEL	C3 ppqq	3	10	STC	37	1	4
JNC LABEL	D2 ppqq	3	10	SUB REG	10010XXX	1	4
JNZ LABEL	C2 ppqq	3	10	SUB M	96	1	7
JP LABEL	F2 ppqq	3	10	SUI DATA	D6 YY	2	7
JPE LABEL	EA ppqq	3	10	XCHG	EB	1	4
JPO LABEL	E2 ppqq	3	10	XRA REG	10101XXX	1	4
J7 LABEL	CA ppqq	3	10	XRA M	AE	1	7
LDA ADDR	3A ppqq	3	13	XRI DATA	EE YY	2	7
LDAX RP	000X1010	1	7	XTHL	E3	1	18
LHD ADDR	2A ppqq	3	16				

ppqq represents four hexadecimal digit memory address
YY represents two hexadecimal data digits
YYYY represents four hexadecimal data digits
X represents an optional binary digit
ddd represents optional binary digits identifying a destination register
sss represents optional binary digits identifying a source register

ACI — ADD WITH CARRY IMMEDIATE TO ACCUMULATOR



To the Accumulator, add the contents of the next program memory byte, and the Carry status
 Suppose $xx = 3A_{16}$, $yy = 7C_{16}$, $C = 0$. After the instruction:

ACI .. 7CH

has executed, the Accumulator will contain B6:

$3A = 00111010$
 $7C = 01111100$
 Carry = 0

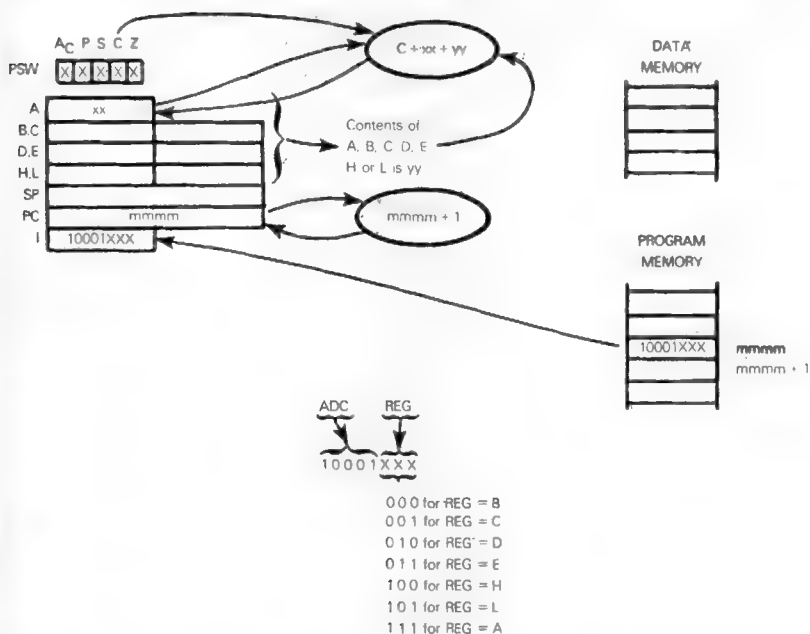
 10110110

Five 1 bits, set P to 0.
 Nonzero result, set Z to 0.
 No carry sets C to 0.
 Carry sets A_C to 1.
 1 sets S to 1.

This is a routine data manipulation instruction.

ADC — ADD REGISTER OR MEMORY WITH CARRY, TO ACCUMULATOR

This instruction takes two forms. First consider a register's contents added to the Accumulator:

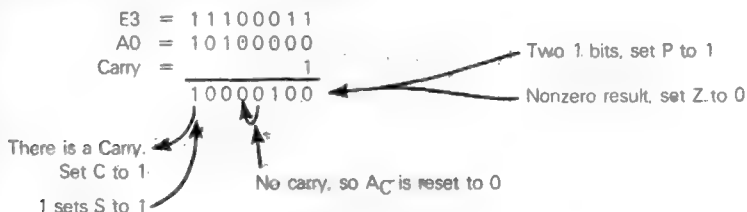


To the Accumulator, add the contents of register A, B, C, D, E, H or L, and add the Carry status.

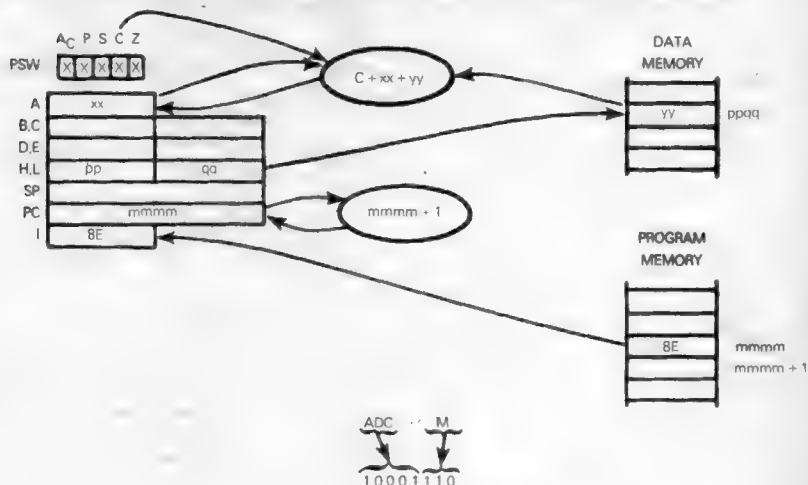
Suppose $xx = E3_{16}$, register E contains $A0_{16}$, $C = 1$. After instruction:

ADC E

has executed, the Accumulator will contain 84_{16} :



The contents of a memory byte may also be added, with Carry, to the Accumulator:



If $xx = E3_{16}$, $yy = A0_{16}$ and $C = 1$, then execution of the instruction:

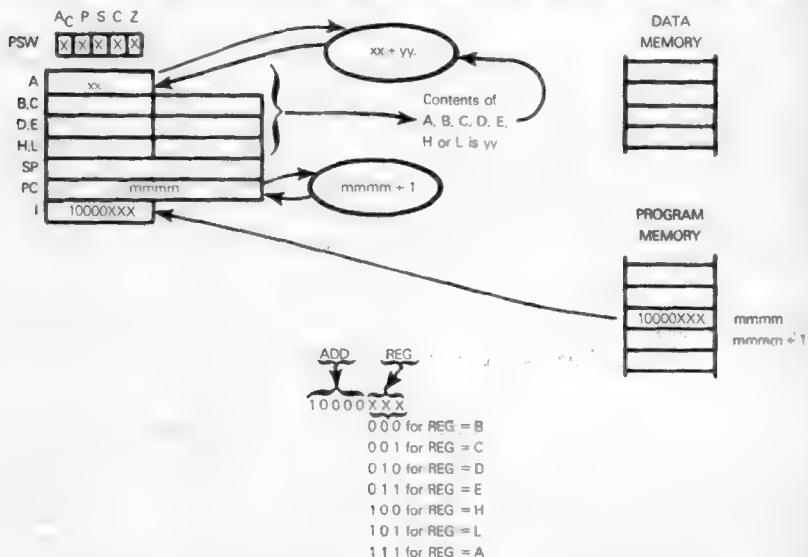
ADC M

generates the same result as execution of the ADC E instruction, which was just described.

The ADC instruction is most frequently used in multibyte addition, for the second and subsequent bytes.

ADD — ADD REGISTER OR MEMORY TO ACCUMULATOR

This instruction takes two forms. First consider a register's contents added to the Accumulator:

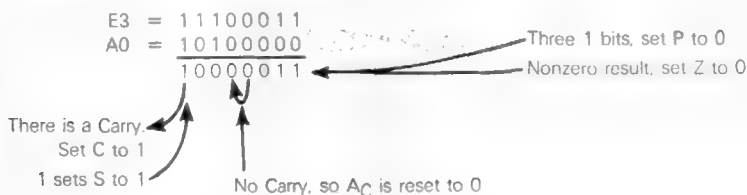


To the Accumulator, add the contents of register A, B, C, D, E, H or L.

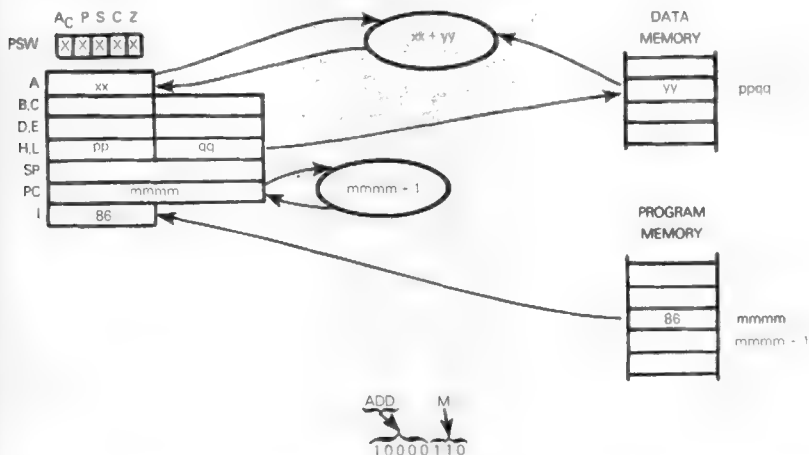
Suppose $xx = E3_{16}$, register E contains $A0_{16}$, $C = 1$. After instruction:

ADD E

has executed, the Accumulator will contain 83_{16} :



The contents of a memory byte may also be added to the Accumulator:



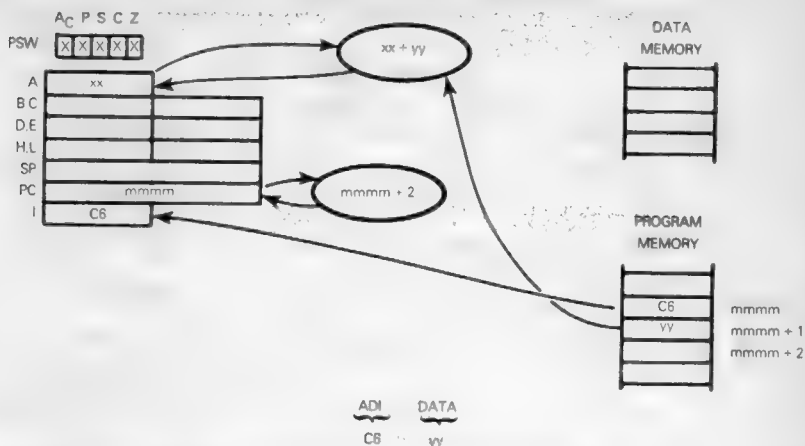
If $xx = E3_{16}$, $yy = A0_{16}$ and $C = 1$, then execution of the instruction:

ADD M

generates the same result as execution of the ADD E instruction, which was just described

ADD is the binary addition instruction used in normal, single-byte operations; it is also the instruction used to add the low order bytes of two multibyte numbers.

ADI — ADD IMMEDIATE TO ACCUMULATOR



To the Accumulator, add the contents of the next program memory byte.

Suppose $xx = 3A_{16}$, $yy = 7C_{16}$, $C = 0$. After the instruction:

ADI 7CH

has executed, the Accumulator will contain B6:

$$\begin{array}{r}
 3A = 00111010 \\
 7C = 01111100 \\
 \hline
 10110110
 \end{array}$$

Five 1 bits, set P to 0

Nonzero result, set Z to 0

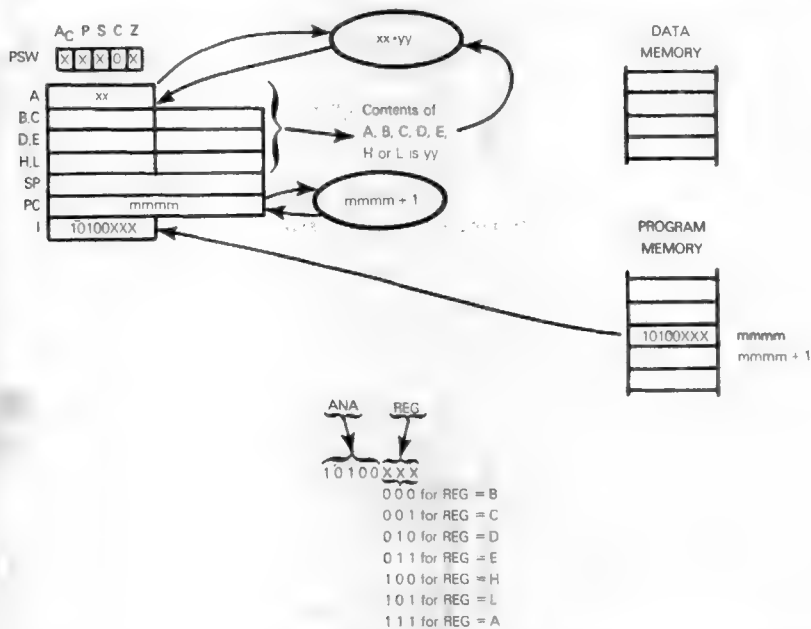
No Carry
sets C to 0
1 sets S to 1

Carry sets A_C to 1

This is a routine data manipulation instruction

ANA — AND REGISTER OR MEMORY WITH ACCUMULATOR

This instruction takes two forms. First consider a register's contents ANDed with the Accumulator:

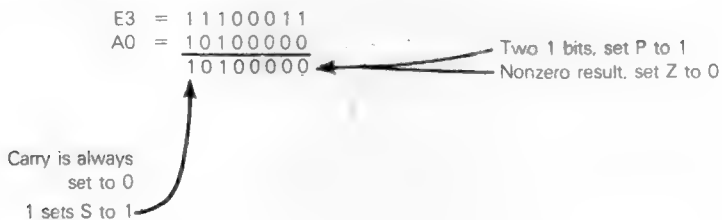


AND the Accumulator with the contents of register A, B, C, D, E, H or L. Save the result in the Accumulator.

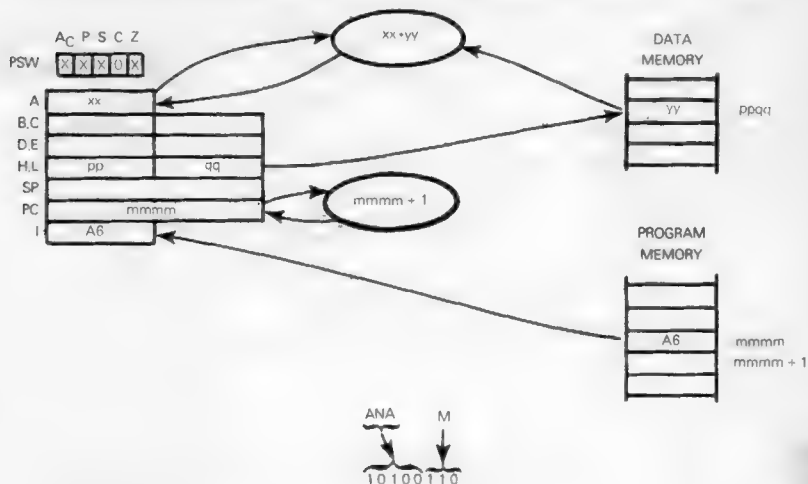
Suppose $xx = E3_{16}$, register E contains $A0_{16}$. After instruction:

ANA \therefore E

has executed, the Accumulator will contain $A0_{16}$:



The contents of a memory byte may also be ANDed to the Accumulator:



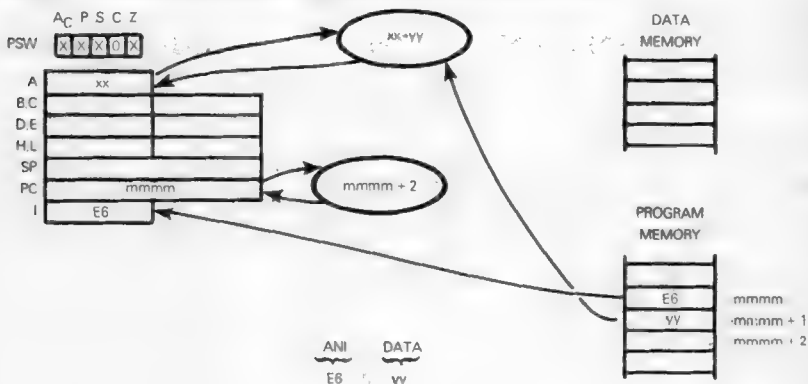
If $xx = E3_{16}$, $yy = A0_{16}$ and $C = 1$, then execution of the instruction:

ANA M

generates the same result as execution of the ANA E instruction, which was just described.

ANA is a frequently used logical instruction.

ANI — AND IMMEDIATE WITH ACCUMULATOR

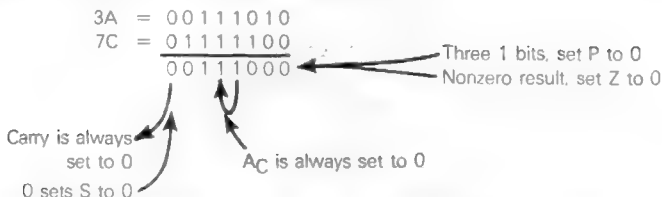


To the Accumulator, AND the contents of the next program memory byte.

Suppose $xx = 3A_{16}$, $yy = 7C_{16}$. After the instruction:

ANI 7CH

has executed, the Accumulator will contain 38_{16} .

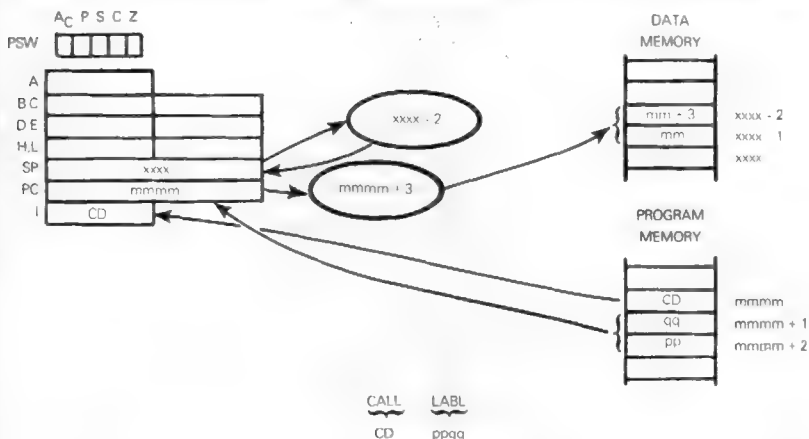


This is a routine logical instruction; it is often used to turn bits "off". For example, the instruction:

ANI 7FH

will unconditionally set the high order Accumulator bit to 0.

CALL — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND



Store the address of the instruction following the **CALL** on the top of the stack, the top of the stack is a data memory byte addressed by the Stack Pointer. Then subtract 2 from the Stack Pointer in order to address the new top of stack. Move the 16-bit address contained in the second and third **CALL** instruction object program bytes to the Program Counter.

Consider the instruction sequence:

CALL SUBR
ANI 7CH

SUBR

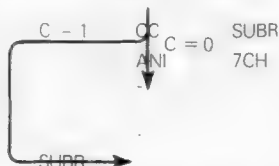
After the **CALL** instruction has executed, the address of the **ANI** instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled **SUBR** will be executed next.

CC — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE CARRY STATUS EQUALS 1

CC
DC

This instruction is identical to the CALL instruction, except that the identified subroutine will be called only if the Carry status equals 1; otherwise, the instruction sequentially following the CC instruction will be executed.

Consider the instruction sequence:



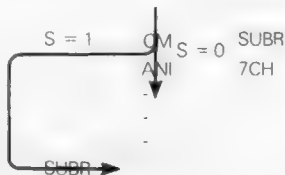
After the CC instruction has executed, if the Carry status does not equal 1, the ANI instruction will be executed. If the Carry status does equal 1, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

CM — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE SIGN STATUS EQUALS 1

CM
FC

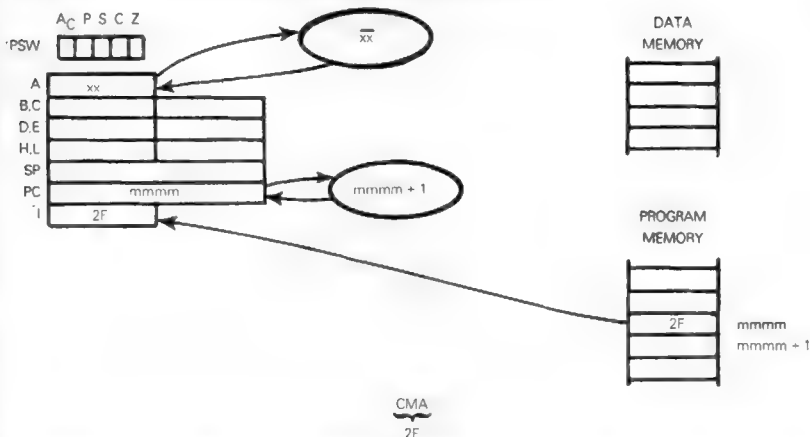
This instruction is identical to the CALL instruction, except that the identified subroutine will be called only if the Sign status equals 1, otherwise, the instruction sequentially following the CM instruction will be executed.

Consider the instruction sequence:



After the CM instruction has executed, if the Sign status does not equal 1, the ANI instruction will be executed. If the Sign status does equal 1, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

CMA — COMPLEMENT THE ACCUMULATOR



Complement the contents of the Accumulator. No other register's contents, or status is effected.

Suppose the Accumulator contains $3A_{16}$. After the instruction:

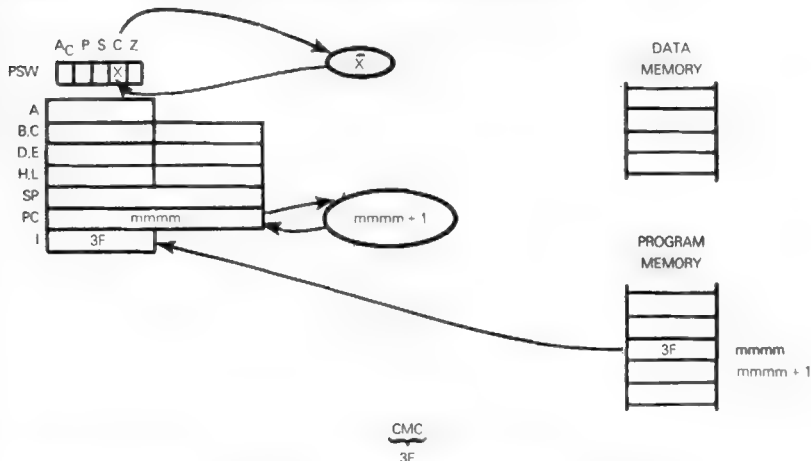
CMA

is executed, the Accumulator will contain $C5_{16}$:

$3A_{16} = 00111010$
Complement = 11000101

This is a routine logical instruction. **Do not use it for binary subtraction.** There are special subtract instructions (SUB and SBB).

CMC — COMPLEMENT THE CARRY STATUS



Complement the Carry status. No other status, or register's contents are effected.

Suppose the Carry status contains 1. After the instruction:

CMC

is executed, the Carry status will contain 0.

This instruction is used to force the Carry status to 0, via the instruction sequence:

STC ;SET CARRY STATUS TO 1
CMC ;COMPLEMENT CARRY STATUS

Note you can set the Carry status to 0 via the instruction:

ANA A

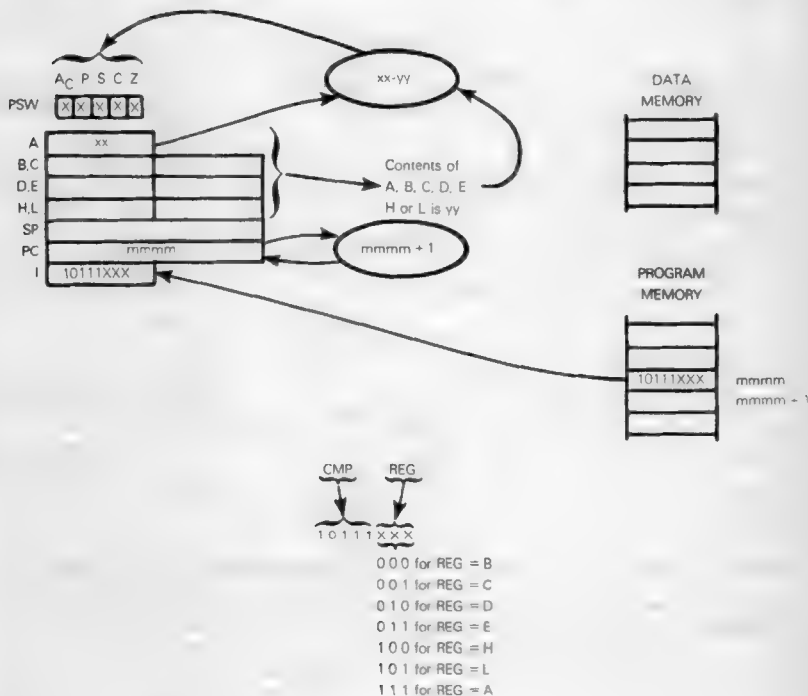
which automatically zeros the Carry status, but does not modify any register's contents, since the Accumulator is ANDed with itself. The:

ORA A

instruction serves the same purpose.

CMP — COMPARE REGISTER OR MEMORY WITH ACCUMULATOR

This instruction takes two forms. First consider a register's contents compared with the Accumulator:



Subtract the contents of register A, B, C, D, H or L from the contents of the Accumulator, treating both numbers as simple binary data. Discard the result, i.e., leave the Accumulator alone, but modify status flags to reflect the result of the subtraction.

Suppose $xx = E3_{16}$, register E contains $A0_{16}$. After instruction:

CMP E

has executed, the Accumulator will still contain E3₁₆, but statuses will be modified as follows:

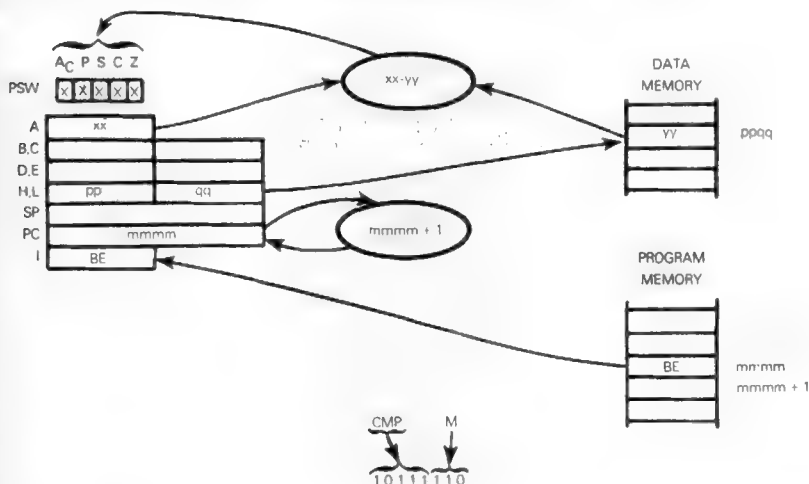
$E3 = 11100011$
 Twos comp of A0 = $\frac{01100000}{01000011}$

Three 1 bits, set P to 0
 Nonzero result, set Z to 0

Carry sets C to 0
 0 sets S to 0
 No Carry so AC is reset to 0

Notice that the resulting Carry is complemented.

The contents of a memory byte may also be compared with the Accumulator:



If $xx = E3_{16}$ and $yy = A0_{16}$, then execution of the instruction:

CMP	M
-----	---

generates the same result as execution of the CMP E instruction, which was just described.

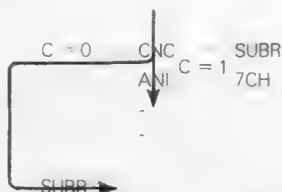
Compare instructions frequently precede conditional Call, Return and Jump instructions. The Compare Immediate (CPI) instruction is more useful than the CMP instruction

CNC — CALL THE SUBROUTINE IDENTIFIED BY THE OPERAND, BUT ONLY IF THE CARRY STATUS EQUALS 0

CNC
D4

This instruction is identical to the CALL instruction, except that the identified subroutine will be called only if the Carry status equals 0; otherwise, the instruction sequentially following the CNC instruction will be executed.

Consider the instruction sequence:



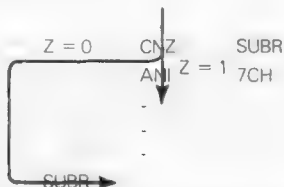
After the CNC instruction has executed, if the Carry status does not equal 0, the ANI instruction will be executed. If the Carry status does equal 0, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

CNZ — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE ZERO STATUS EQUALS 0

CNZ
C4

This instruction is identical to the CALL instruction, except that the identified subroutine will be called only if the Zero status equals 0; otherwise, the instruction sequentially following the CNZ instruction will be executed.

Consider the instruction sequence:



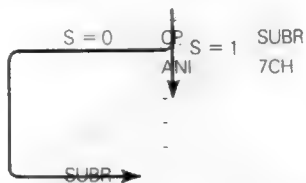
After the CNZ instruction has executed, if the Zero status does not equal 0, the ANI instruction will be executed. If the Zero status does equal 0, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

CP — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE SIGN STATUS EQUALS 0

CP
F4

This instruction is identical to the CALL instruction, except that the identified subroutine will be called only if the Sign status equals 0; otherwise, the instruction sequentially following the CP instruction will be executed.

Consider the instruction sequence:



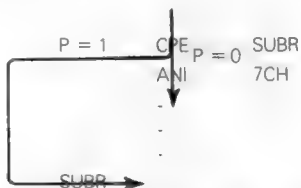
After the CP instruction has executed, if the Sign status does not equal 0, the ANI instruction will be executed. If the Sign status does equal 0, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

CPE — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE PARITY STATUS EQUALS 1

CPE
EC

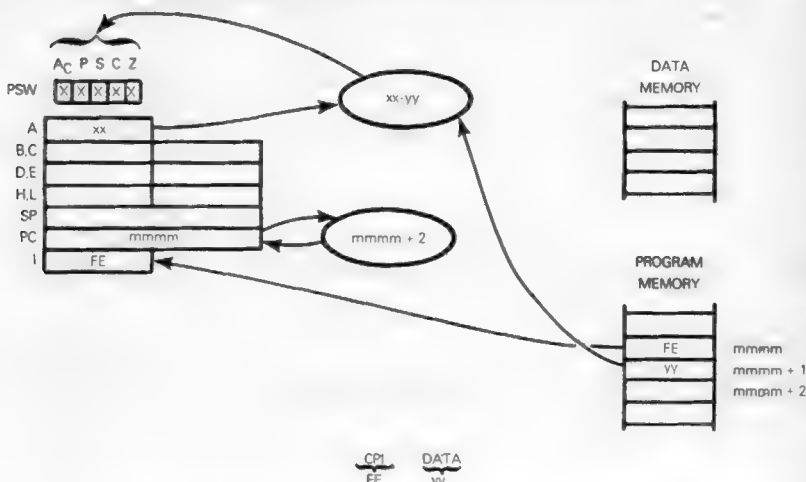
This instruction is identical to the CALL instruction, except that the identified subroutine will be called only if the Parity status equals 1, otherwise, the instruction sequentially following the CPE instruction will be executed.

Consider the instruction sequence:



After the CPE instruction has executed, if the Parity status does not equal 1, the ANI instruction will be executed. If the Parity status does equal 1, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

CPI — COMPARE ACCUMULATOR CONTENTS WITH IMMEDIATE DATA



Subtract the contents of the second object code byte from the contents of the Accumulator, treating both numbers as simple, binary data. Discard the result, i.e., leave the Accumulator alone, but modify the status flags to reflect the result of the subtraction.

Suppose $xx = E3_{16}$ and the second byte of the CPI instruction object code contains $A0_{16}$. After instruction:

CPI $\cdot A0H$

has executed, the Accumulator will still contain $E3_{16}$, but statuses will be modified as follows:

$E3 = 11100011$
 Twos comp. of $A0 = 01100000$
 01000011

Three 1 bits, set P to 0
 Nonzero result, set Z to 0
 Carry sets C to 0
 0 sets S to 0
 No carry, so A_C is reset to 0

Notice that the resulting Carry is complemented.

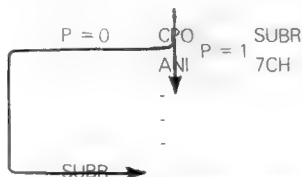
This is the instruction most frequently used to set statuses prior to execution of a conditional Call, Return or Jump instruction.

CPO — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE PARITY STATUS EQUALS 0

CPO
 E4

This instruction is identical to the CALL instruction, except that the identified subroutine will be called only if the Parity status equals 0. otherwise, the instruction sequentially following the CPO instruction will be executed

Consider the instruction sequence:



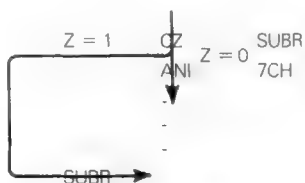
After the CPO instruction has executed, if the Parity status does not equal 0, the ANI instruction will be executed. If the Parity status does equal 0, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

CZ — CALL THE SUBROUTINE IDENTIFIED IN THE OPERAND, BUT ONLY IF THE ZERO STATUS EQUALS 1

CZ
CC

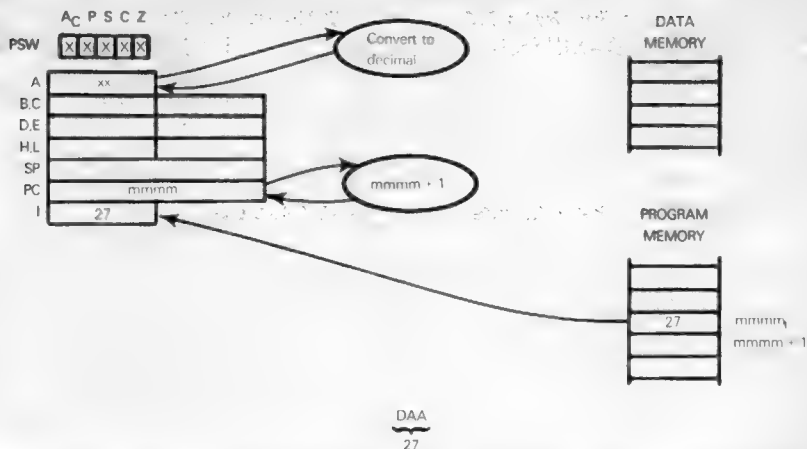
This instruction is identical to the CALL instruction, except that the identified subroutine will be called only if the Zero status equals 1; otherwise, the instruction sequentially following the CZ instruction will be executed.

Consider the instruction sequence:



After the CZ instruction has executed, if the Zero status does not equal 1, the ANI instruction will be executed. If the Zero status does equal 1, the address of the ANI instruction is saved at the top of the stack. The Stack Pointer is decremented by 2. The instruction labeled SUBR will be executed next.

DAA — DECIMAL ADJUST ACCUMULATOR



Convert the contents of the Accumulator to its binary coded decimal form. This instruction should be used only after adding two BCD numbers, i.e., look upon ADD DAA or ADC DAA or SUB DAA or SBB DAA as compound, decimal arithmetic instructions which operate on BCD source to generate BCD answers.

Suppose the Accumulator contains 39_{16} and the B register contains 47_{16} . After the instructions:

```
ADD B
DAA
```

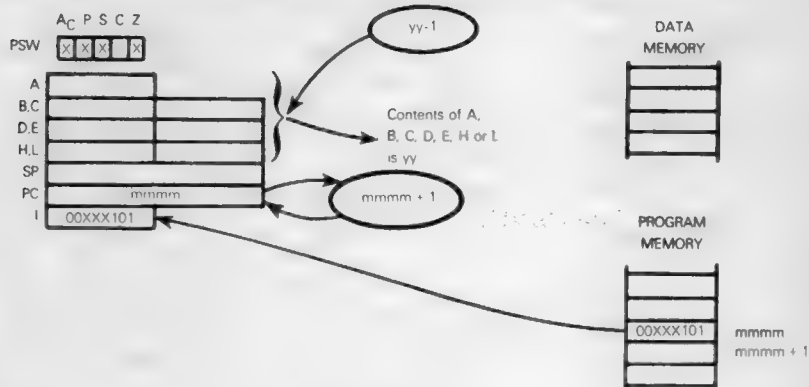
have executed, the Accumulator will contain 86_{16} , not 80_{16} .

The DAA instruction modifies all status flags, but only the Carry status is significant. Consider other statuses to have been destroyed.

A P S C Z



DCR — DECREMENT REGISTER OR MEMORY CONTENTS



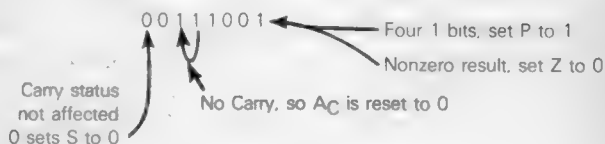
0 00 for REG = B
 0 01 for REG = C
 0 10 for REG = D
 0 11 for REG = E
 1 00 for REG = H
 1 01 for REG = L
 1 11 for REG = A

Subtract 1 from the contents of the specified register.

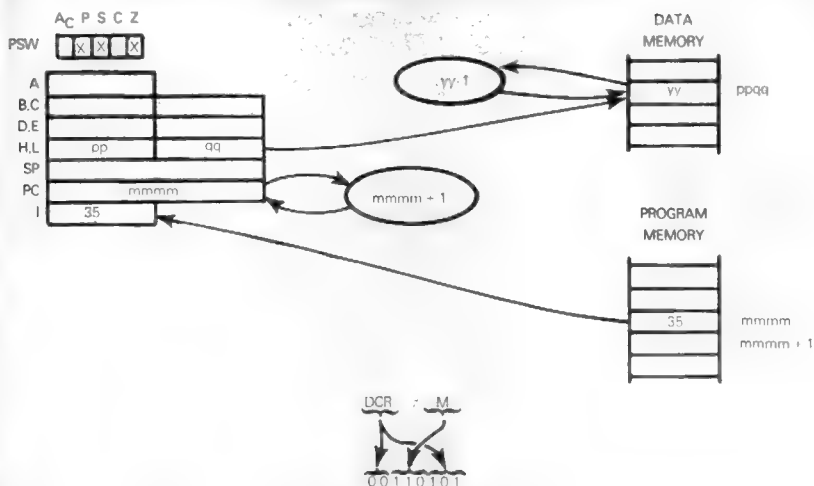
Suppose the C register contains $3A_{16}$. After instruction:

DCR C

has executed, the C register will contain 39_{16} :



The contents of a read/write memory byte may also be decremented:



Suppose HL contains 3714_{16} . Then execution of the instruction:

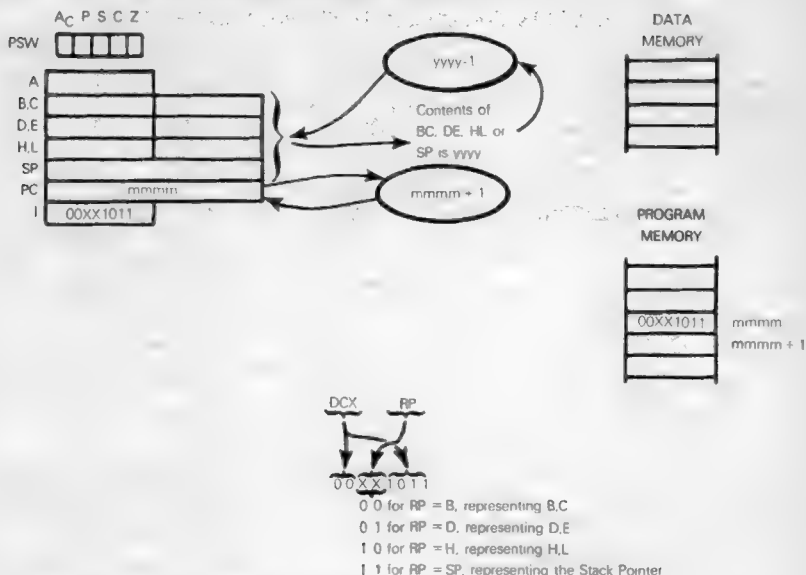
DCR \rightarrow M

subtracts 1 from the contents of the memory byte with address 3714_{16} . Status flags are modified as described for the DCR C instruction.

The DCR instruction is used in iterative instruction loops that use a counter with a value of 256 or less. This is the typical loop form:

MVI	REG, DATA	:LOAD INITIAL COUNTER VALUE
-		:FIRST INSTRUCTION OF LOOP
-		
-		
-		
DCR	REG	:DECREMENT COUNTER
JNZ	LOOP	:RETURN IF NOT ZERO

DCX — DECREMENT REGISTER PAIR



Subtract 1 from the 16-bit value contained in the specified register pair.

Suppose the Stack Pointer contains $2F7A_{16}$. After instruction:

DCX SP

has executed, the Stack Pointer will contain $2F79_{16}$.

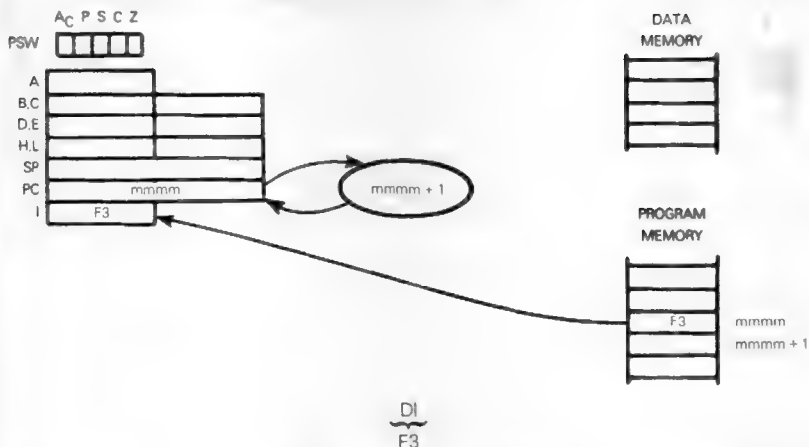
The DCX instruction does not modify any status flags, and this is a defect in the 8080 instruction set. Whereas the DCR instruction is used in iterative instruction loops that use a counter with a value of 256 or less, the DCX instruction must be used if the counter value is more than 256. Since the DCX instruction sets no status flags, other instructions must be added simply to test for a zero result. This is a typical loop form:

```

LXI    D,DATA    ;LOAD INITIAL 16-BIT COUNTER VALUE
LOOP   :FIRST INSTRUCTION OF LOOP

      DCX    D    ;DECREMENT COUNTER
      MOV    A,D    ;TO TEST FOR ZERO, MOVE D TO A
      ORA    E    ;THEN OR A WITH E
      JNZ    LOOP   ;RETURN IF NOT ZERO
    
```

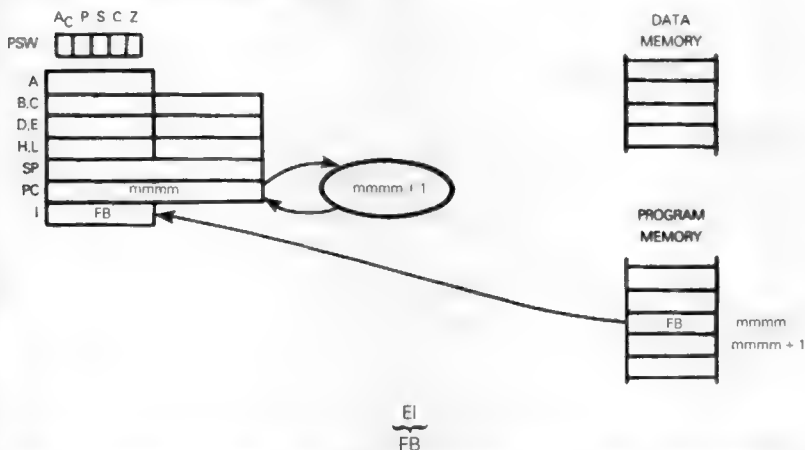
DI — DISABLE INTERRUPTS



After this instruction has been executed the INTE signal is output low by the 8080 CPU, and no interrupt requests will be acknowledged. No registers or status flags are effected.

Remember that when an interrupt is acknowledged, interrupts are automatically disabled.

EI — ENABLE INTERRUPTS



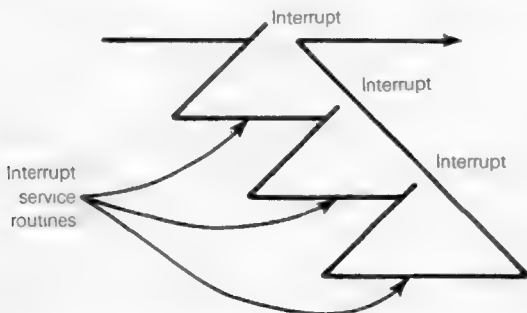
When this instruction is executed, interrupts are enabled, but not until one more instruction executes.

Most interrupt service routines end with the two instructions:

```
EI      ;ENABLE INTERRUPTS
RET     ;RETURN TO INTERRUPTED PROGRAM
```

If interrupts are processed serially, then for the entire duration of the interrupt service routine, all interrupts are disabled — which means that in a multi-interrupt application, there is a significant possibility for one or more interrupts to be pending when any interrupt service routine completes execution.

If interrupts were acknowledged as soon as the EI instructions had executed, then the Return instruction would not be executed. Under these circumstances returns would stack up one on top of the other — and unnecessarily consume stack memory space. This may be illustrated as follows:



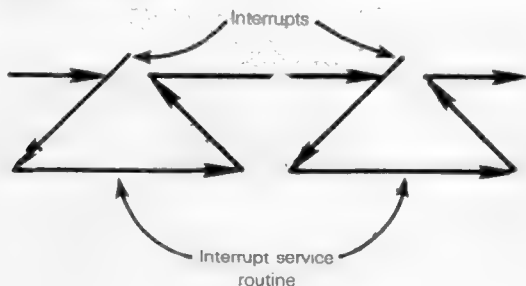
By inhibiting interrupts for one more instruction following execution of EI, the 8080 CPU insures that the RET instruction gets executed in the sequence:

```

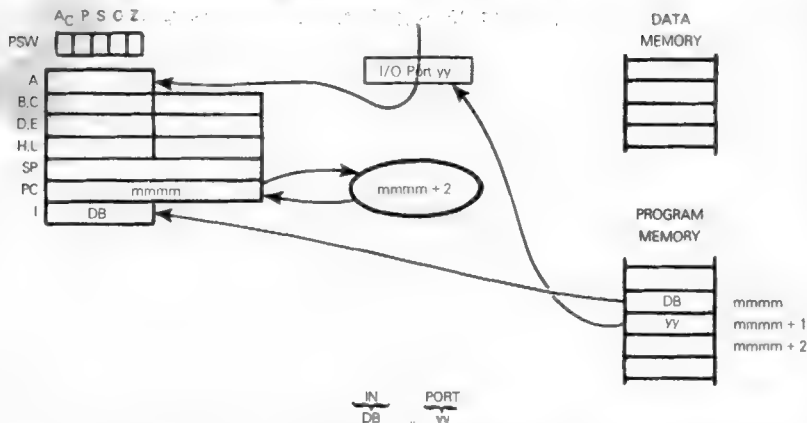
EI      : ENABLE INTERRUPTS
RET     : RETURN FROM INTERRUPT

```

It is not uncommon for interrupts to be kept disabled while an interrupt service routine is executing. Interrupts are processed serially:



IN — INPUT TO ACCUMULATOR



Load a byte of data into the Accumulator from the I/O port identified by the second IN instruction object code byte.

Suppose 36₁₆ is held in the buffer of I/O port 36₁₆. After the instruction:

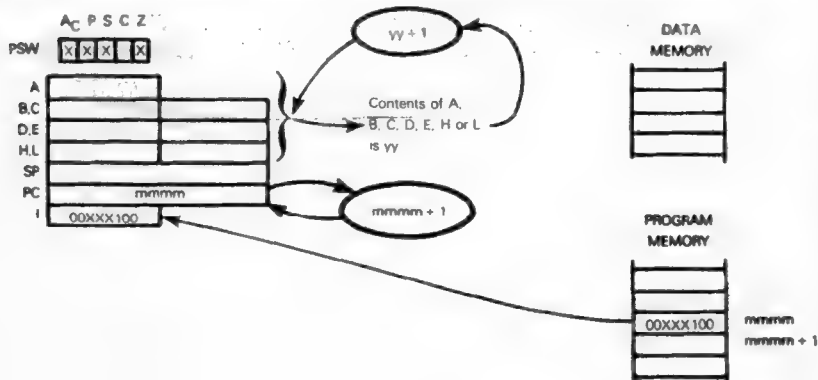
IN 1AH

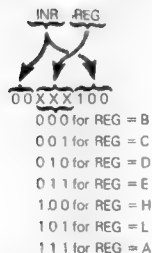
has executed, the Accumulator will contain 1A₁₆.

The IN instruction does not affect any statuses.

Use of the IN instruction is very hardware dependent. Valid I/O port addresses are determined by the way in which I/O logic has been implemented. It is also possible to design a microcomputer system that accesses external logic using memory reference instructions with specific memory addresses.

INR — INCREMENT REGISTER OR MEMORY CONTENTS



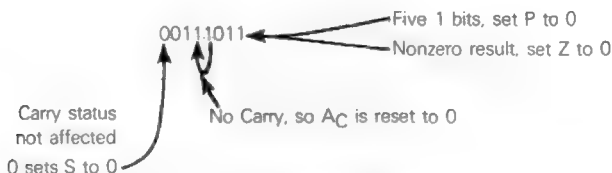


Add 1 to the contents of the specified register.

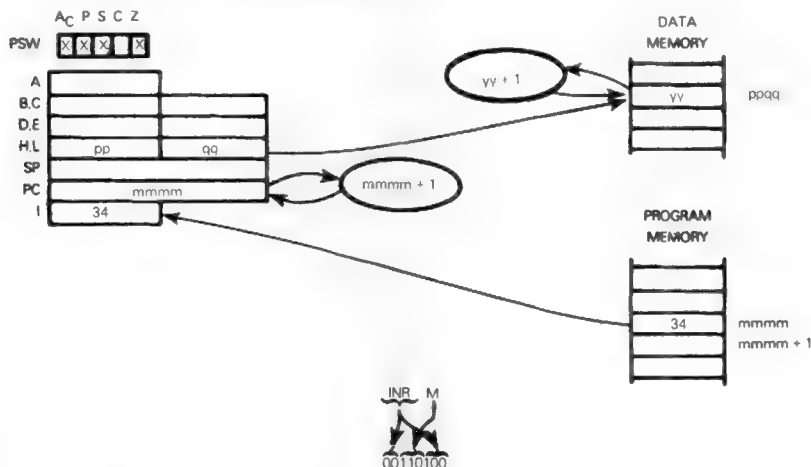
Suppose the C register contains $3A_{16}$. After instruction:

INR C

has executed, the C register will contain $3B_{16}$:



The contents of a read/write memory byte may also be incremented:



Suppose HL contains 3714_{16} . Then execution of the instruction:

INR M

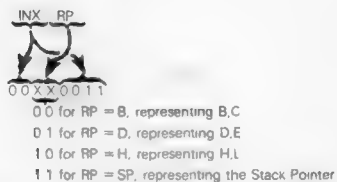
adds 1 to the contents of the memory byte with address 3714_{16} . Status flags are modified as described for the INR C instruction.

	MVI	REG,DATA	:LOAD COMPLEMENT OF INITIAL COUNTER VALUE
LOOP	JNZ	END OF LOOP	:FIRST INSTRUCTION OF LOOP

INR	REG	DECREMENT COUNTER
JNZ	LOOP	RETURN IF NOT ZERO

Diagram illustrating the instruction format and execution flow for the MOV instruction:

- Instruction Format:**
 - Opcode: A_C_P_S_C_Z (5 bits)
 - Register Field: A, B.C, D.E, H.L (4 bits)
 - Address Field: SP, PC, I (8 bits)
 - Address Field Content: mmmm, 00XX0011
- Execution Flow:**
 - The register field (B.C, D.E, H.L, or SP) selects the source register containing the value **yyyy**.
 - The address field (SP, PC, or I) selects the destination register containing the value **mmmm**.
 - The source register value **yyyy** is incremented to **yyyy + 1**.
 - The destination register value **mmmm** is incremented to **mmmm + 1**.
 - The data is moved from the source register to the destination register.
- Memory Structures:**
 - DATA MEMORY:** Contains the source register value **yyyy**.
 - PROGRAM MEMORY:** Contains the destination register value **mmmm** and the instruction address **00XX0011**.



Suppose the D and E registers contain 2F7A₁₆. After instruction:

INX D

has executed, the D and E registers will contain 2F7B₁₆.

The INX instruction does not modify any status flags, and this is a defect in the 8080 instruction set. Whereas the DCR instruction is used in iterative instruction loops that use a counter with a value of 256 or less, the INX instruction must be used if the counter value is more than 256. Since the INX instruction sets no status flags, other instructions must be added simply to test for a zero result. This is a typical loop form:

```

LXI 0000H,DATA      ;LOAD COMPLEMENT OF INITIAL 16-BIT COUNTER
                     ;VALUE
LOOP 0000H,0000H     ;FIRST INSTRUCTION OF LOOP

```

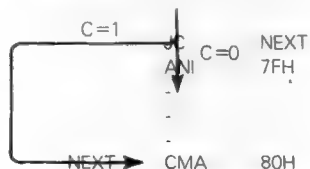
INX	D	: INCREMENT COUNTER
MOV	A,D	: TO TEST FOR ZERO, MOVE D TO A,
ORA	E	: THEN OR A WITH E
JNZ	LOOP	: RETURN IF NOT ZERO

JC — JUMP IF CARRY

JC
DA

This instruction is identical to the JMP instruction except that the jump is only executed if the Carry status equals 1, otherwise the next instruction is executed.

In the following instruction sequence:



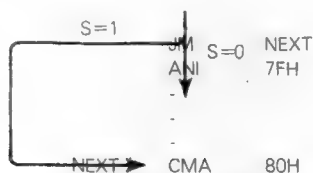
after the JC instruction, the CMA instruction is executed if the Carry status equals 1. The ANI instruction is executed if the Carry status equals 0.

JM — JUMP IF MINUS

JM
FA

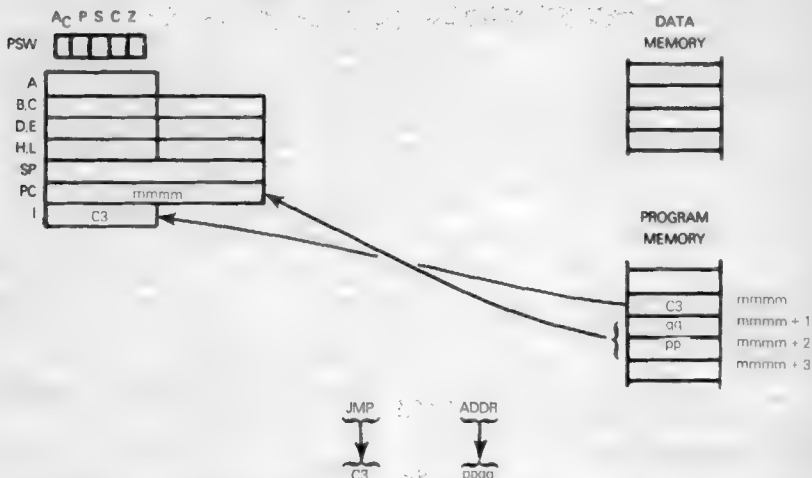
This instruction is identical to the JMP instruction except that the jump is only executed if the Sign status equals 1; otherwise the next instruction is executed.

In the following instruction sequence:



after the JM instruction, the CMA instruction is executed if the Sign status equals 1. The ANI instruction is executed if the Sign status equals 0.

JMP — JUMP TO THE INSTRUCTION IDENTIFIED IN THE OPERAND



Load the contents of the Jump instruction object code second and third bytes into the Program Counter, this becomes the memory address for the next instruction to be executed. The previous Program Counter contents are lost.

In the following instruction sequence:

```

JMP     NEXT
ANI     7FH
.
.
.

```

```

NEXT    CMA     80H

```

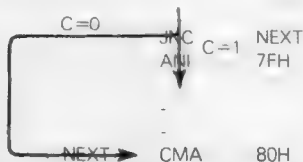
after the JMP instruction, the CMA instruction will be executed. The ANI instruction will never be executed unless a Jump instruction somewhere else in the instruction sequence jumps to this instruction.

JNC — JUMP IF NO CARRY

JNC
D2

This instruction is identical to the JMP instruction except that the jump is only executed if the Carry status equals 0; otherwise the next instruction is executed.

In the following instruction sequence:



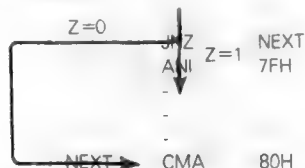
after the JNC instruction, the CMA instruction is executed if the Carry status equals 0. The ANI instruction is executed if the Carry status equals 1.

JNZ — JUMP IF NOT ZERO

JNZ
C2

This instruction is identical to the JMP instruction except that the jump is only executed if the Zero status equals 0; otherwise the next instruction is executed.

In the following instruction sequence:



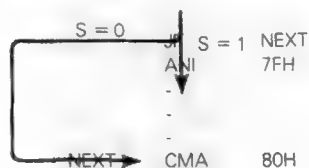
after the JNZ instruction, the CMA instruction is executed if the Zero status equals 0. The ANI instruction is executed if the Zero status equals 1.

JP — JUMP IF PLUS

JP
F2

This instruction is identical to the JMP instruction except that the jump is only executed if the Sign status equals 0; otherwise the next instruction is executed.

In the following instruction sequence:



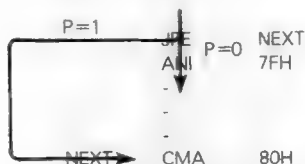
after the JP instruction, the CMA instruction is executed if the Sign status equals 0. The ANI instruction is executed if the Sign status equals 1.

JPE — JUMP IF PARITY EVEN

JPE
EA

This instruction is identical to the JMP instruction except that the jump is only executed if the Parity status equals 1; otherwise the next instruction is executed.

In the following instruction sequence:



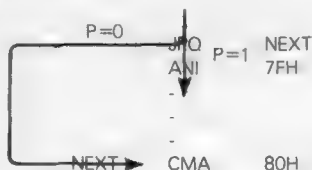
after the JPE instruction, the CMA instruction is executed if the Parity status equals 1. The ANI instruction is executed if the Parity status equals 0.

JPO — JUMP IF PARITY ODD

JPO
E2

This instruction is identical to the JMP instruction except that the jump is only executed if the Parity status equals 0; otherwise the next instruction is executed.

In the following instruction sequence:



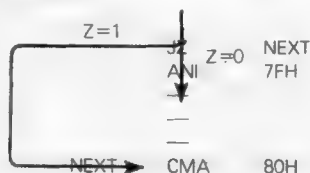
after the JPO instruction, the CMA instruction is executed if the Parity status equals 0. The ANI instruction is executed if the Parity status equals 1.

JZ — JUMP IF ZERO

JZ
CA

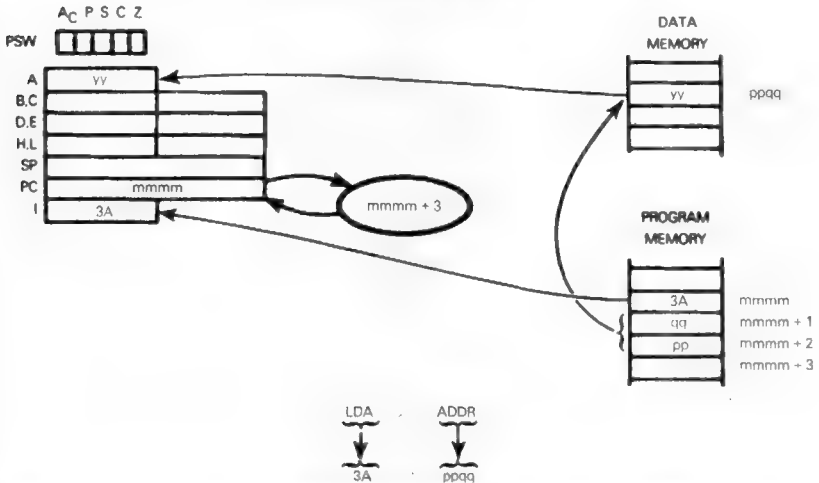
This instruction is identical to the JMP instruction except that the jump is only executed if the Zero status equals 1; otherwise the next instruction is executed.

In the following instruction sequence:



after the JZ instruction, the CMA instruction is executed if the Zero status equals 1. The ANI instruction is executed if the Zero status equals 0.

LDA — LOAD ACCUMULATOR FROM MEMORY USING DIRECT ADDRESSING



Load into the Accumulator the contents of the memory byte addressed directly by the second and third bytes of the LDA instruction object code.

Suppose memory byte $084A_{16}$ contains $3A_{16}$. After the instruction:

```

LABEL EQU 084AH
-
-
-
LDA LABEL
    
```

has executed, the Accumulator will contain $3A_{16}$.

Remember, EQU is an Assembler Directive, it is not an instruction; it tells the Assembler to use the 16-bit value $084A_{16}$ wherever LABEL appears.

The instruction:

```
LDA LABEL
```

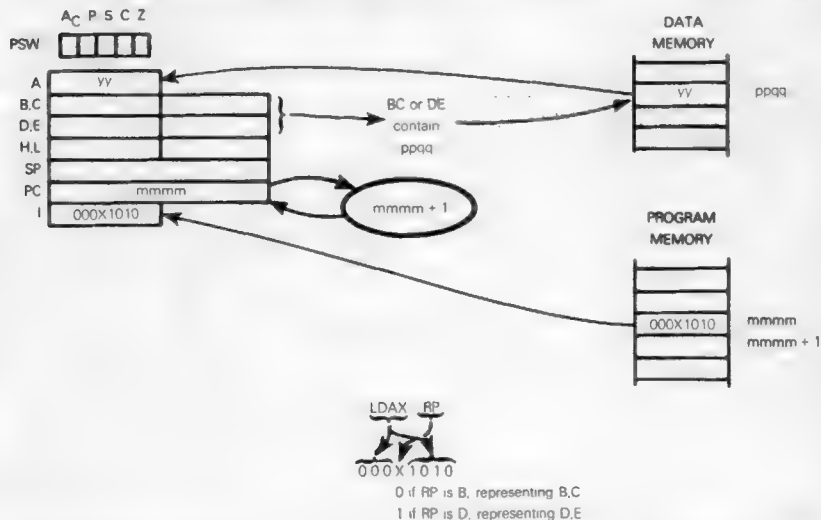
is equivalent to the two instructions:

```

LXI H,LABEL
MOV A,M
    
```

When you are loading a single data value from memory, the LDA instruction is preferred; it uses one instruction and three object program bytes to do what the LXI MOV combination does in two instructions and four object program bytes. Also, the LXI MOV combination uses the H and L registers; the LDA instruction does not.

LDAX — LOAD ACCUMULATOR FROM MEMORY LOCATION ADDRESSED BY REGISTER PAIR



Load into the Accumulator the contents of the memory byte addressed by the BC, or DE register pair.

Suppose the B register contains 08_{16} , the C register contains $4A_{16}$ and memory byte $084A_{16}$ contains $3A_{16}$. After the instruction:

LDAX B

has executed, the Accumulator will contain $3A_{16}$.

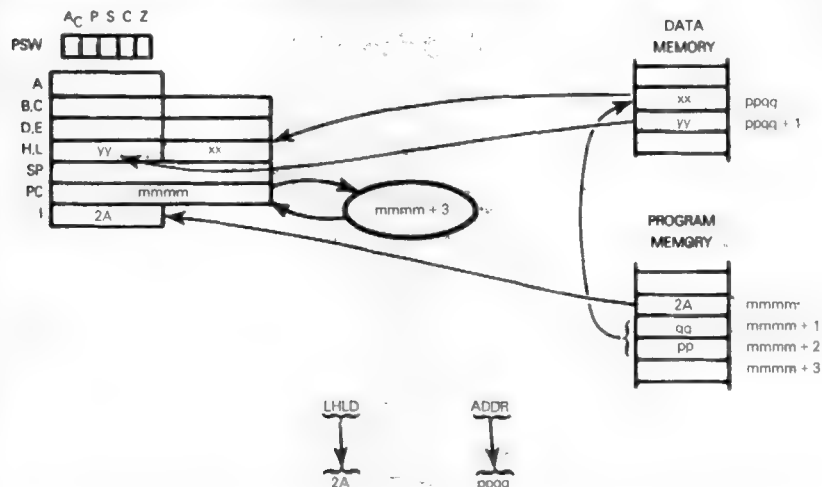
Note that there is no LDAX H instruction since this is identical to a MOV A,M instruction

Normally the LDAX and LXI instructions will be used together, since the LXI instruction loads a 16-bit address into the BC or DE registers, as follows:

LXI B, $084A_{16}$
LDAX B

Notice that the LDAX instruction will only load data into the Accumulator, whereas the MOV instruction will load data into any register.

LHLD — LOAD H AND L REGISTERS DIRECT



The second and third object code bytes provide the memory address of a data byte, the contents of which will be loaded into the L register. The contents of the next sequential data byte is loaded into the H register.

Suppose memory byte $084A_{16}$ contains $3A_{16}$ and memory byte $084B_{16}$ contains $2C_{16}$. After instruction:

```

LABEL EQU 084AH
-
-
-
LHLD LABEL
    
```

has executed, the H register will contain $2C_{16}$ and the L register will contain $3A_{16}$.

Remember, EQU is an Assembler Directive, it is not an instruction; it tells the Assembler to use the 16-bit value $084A_{16}$ wherever LABEL appears.

The LHLD instruction is a direct addressing version of the LXI H,DATA instruction. For example, the instruction:

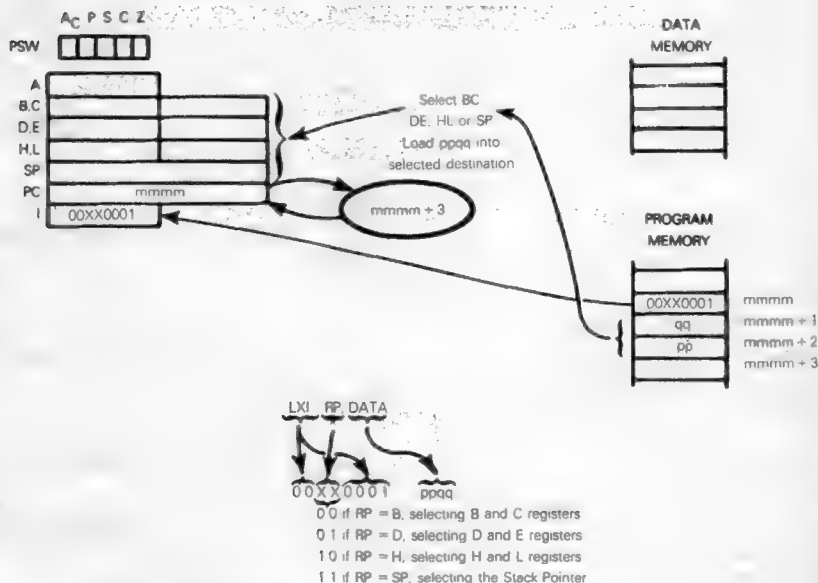
```
LXI H,2C3AH
```

will also load $2C_{16}$ into the H register and $3A_{16}$ into the L register.

For 16-bit data that never changes, use LXI H,DATA instead of LHLD ADDR.

Remember, if ADDR directly addresses a byte of read/write memory, you can change the value that will be loaded into the H and L registers by an LHLD instruction. To do this, simply write the new value into ADDR and ADDR + 1.

LXI — LOAD A 16-BIT VALUE, IMMEDIATE, INTO A REGISTER PAIR



Load into the selected register pair the contents of the second and third object code bytes.

After the instruction:

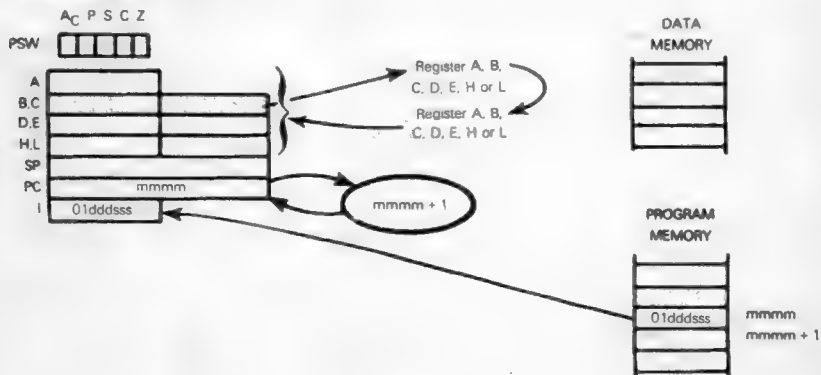
LXI SP, 217AH

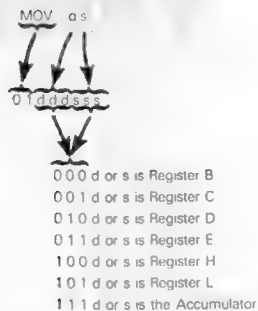
has executed, the Stack Pointer will contain 217A₁₆.

LXI is the instruction most frequently used to load addresses into a register pair.

MOV — MOVE DATA

This instruction takes two forms. Consider first the contents of one register being moved to another register.





Move the contents of any register to any register. For example,

MOV A,B

moves the contents of the B register to the Accumulator.

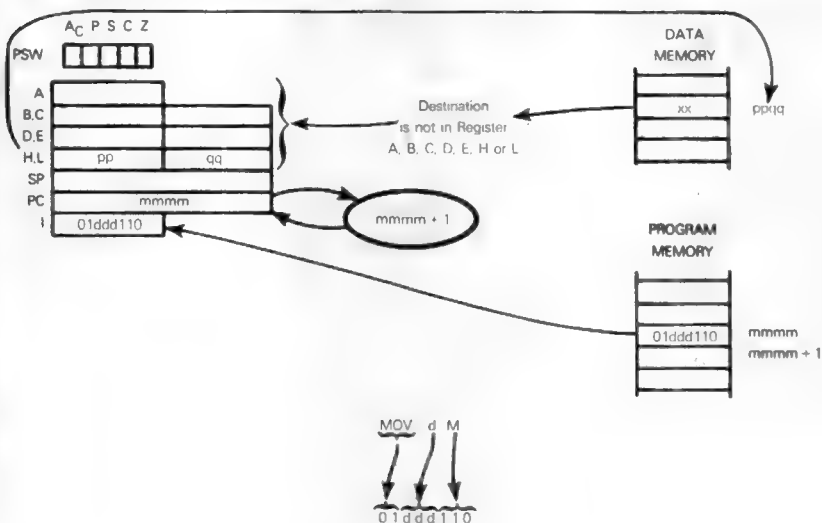
MOV L,D

moves the contents of the D register to the L register.

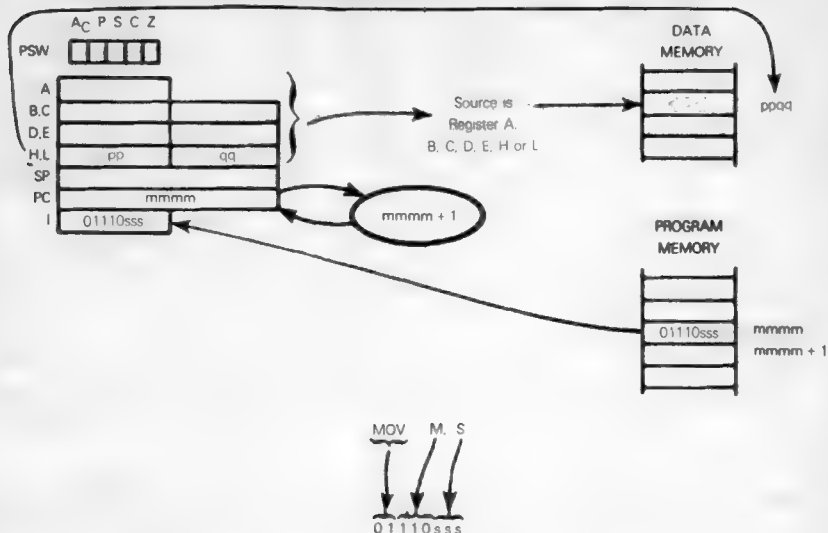
MOV C,C

does nothing, since the C register has been specified as both the source and the destination.

A memory byte may also be the data source:



Or a memory byte may be the data destination:



In either case, ddd or sss is interpreted as for a register-to-register move.

Thus the instruction:

MOV M,A

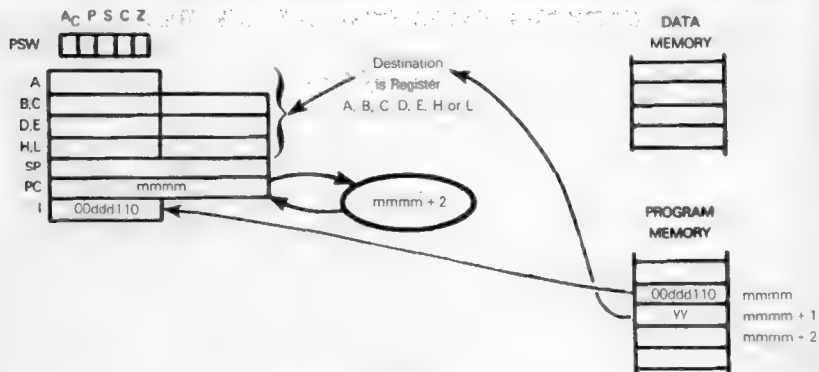
moves the Accumulator contents to the read/write memory byte addressed by the H and L registers. The instruction:

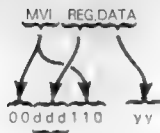
MOV L,M

moves the contents of the memory byte addressed by the H and L registers into the L register.

The Move instruction in its various forms is the most frequently used of the 8080 instructions.

MVI — LOAD DATA IMMEDIATE INTO REGISTER OR MEMORY





0 0 0 for REG = B
 0 0 1 for REG = C
 0 1 0 for REG = D
 0 1 1 for REG = E
 1 0 0 for REG = H
 1 0 1 for REG = L
 1 1 1 for REG = A

Move the contents of the second object code byte to one of the registers.

When the instruction:

MVI A, 2AH

is executed, 2A₁₆ is loaded into the Accumulator. The instruction:

MVI H, 03H

loads 03₁₆ into the H register.

Load data immediate into register instructions are very frequently used in 8080 programs

Notice that the LXI instruction is equivalent to two MVI instructions; for example,

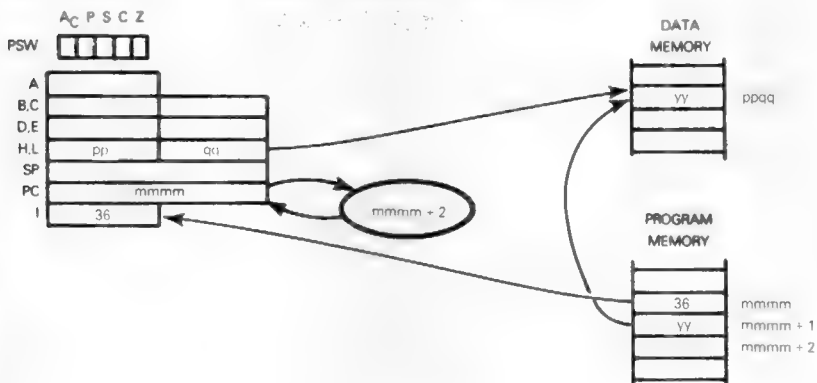
LXI H, 032AH

is equivalent to:

MVI H, 03H

MVI L, 2AH

Data may also be loaded immediately into a byte of read/write memory:



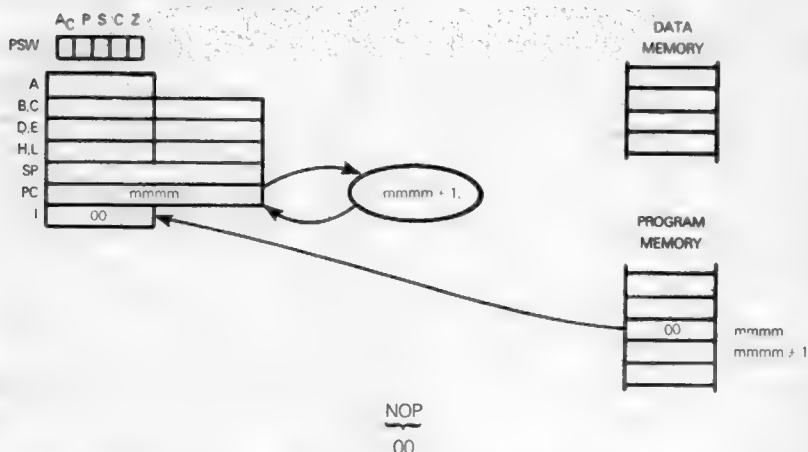
Suppose the H register contains 03_{16} and the L register contains $2A_{16}$; then when the instruction:

MVI M,2CH

is executed, $2C_{16}$ will be loaded into memory byte $032A_{16}$.

The load immediate into memory instruction (MVI M,DATA) is used much less than the load immediate into register instruction (MVI REG,DATA).

NOP — NO OPERATION



Nothing happens when this instruction is executed; it is present for two reasons:

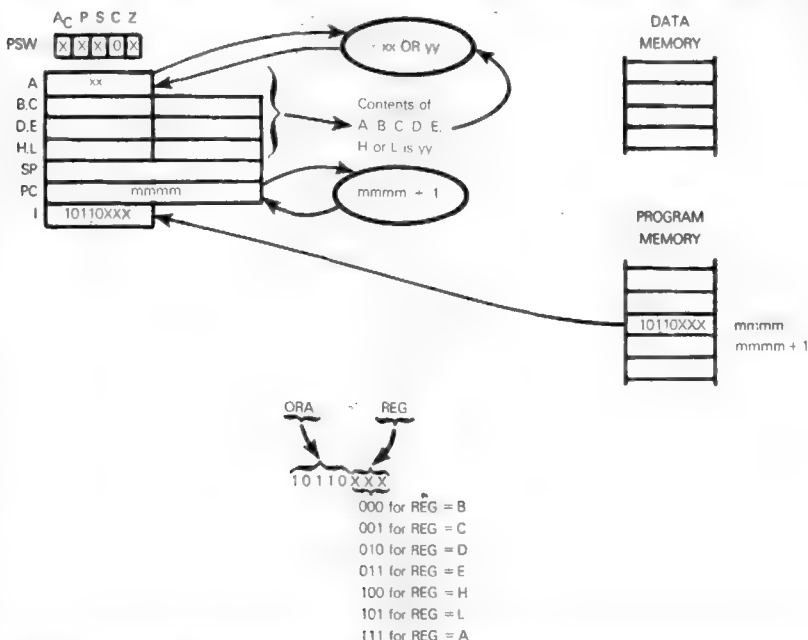
- 1) A program error that fetches an object code from non-existent memory will fetch 00. It is a good idea to insure that the commonest program error will do nothing.
- 2) The NOP instruction allows you to give a label to an object program byte:

HERE NOP

- 3) To fine tune delay times. Each NOP instruction adds four clock cycles to a delay.

NOP is not a very useful, or frequently used instruction.

ORA — OR REGISTER OR MEMORY WITH ACCUMULATOR



Logically OR the contents of the Accumulator with the contents of any register. Store the result in the Accumulator.

Suppose $xx = E3_{16}$, register E contains $A8_{16}$. After instruction:

ORA E

has executed, the Accumulator will contain EB_{16} .

E3 = 11100011

A8 = 10101000

11101011

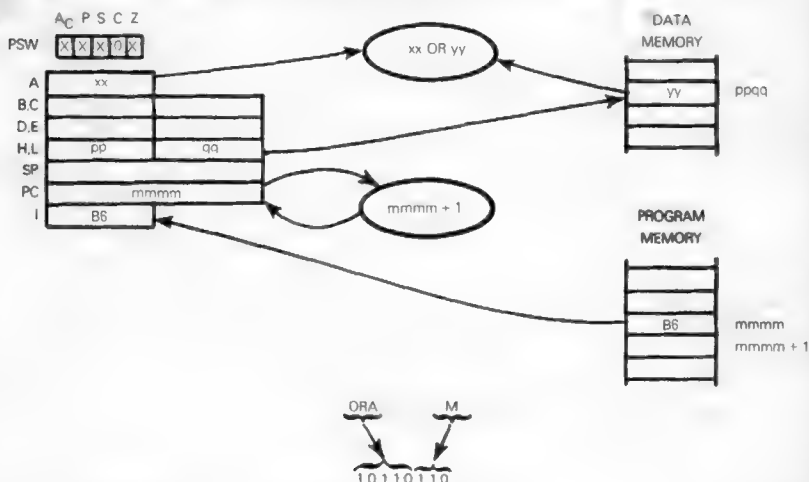
Six 1 bits, set P to 1

Nonzero result, set Z to 0

Carry is always set to 0

1 sets S to 1

The contents of a memory byte may also be ORed with the Accumulator:



If $xx = E3_{16}$ and $yy = A8_{16}$, then execution of the instruction:

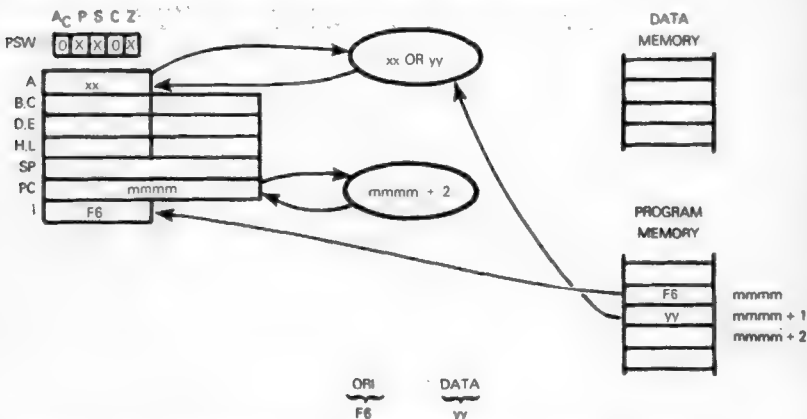
ORA M

generates the same result as execution of the ORA E instruction, which was just described.

ORA is not used as frequently as the OR immediate (ORI) instruction.

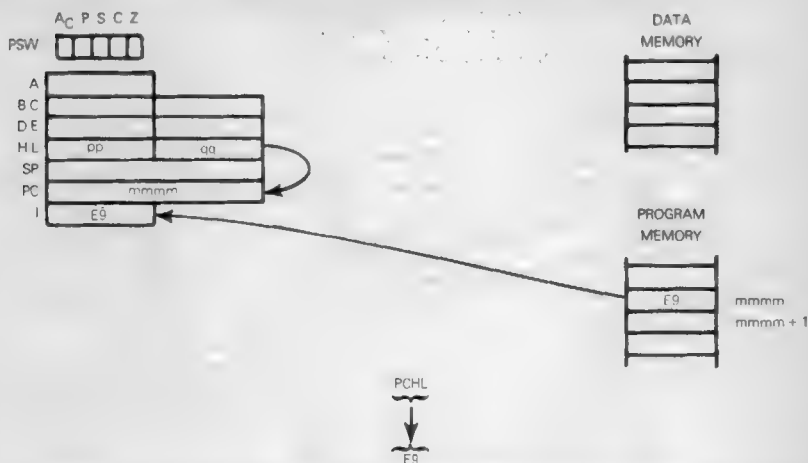
Note that ORing the Accumulator with itself (ORA A) allows you to clear the Carry status; this instruction is also used to set statuses following INX and DCX instructions.

ORI — OR IMMEDIATE WITH ACCUMULATOR



OR the Accumulator with the contents of the second-instruction object code byte.

PCHL — JUMP TO ADDRESS SPECIFIED BY HL



The contents of the H and L registers are moved to the Program Counter, therefore an implied addressing jump is performed.

The instruction sequence:

```
LXI    H,ADDR
PCHL
```

has exactly the same net effect as the single instruction:

```
JMP    ADDR
```

Both specify that the instruction with label ADDR is to be executed next.

The PCHL instruction is useful when you want to increment a return address for a subroutine that has multiple returns.

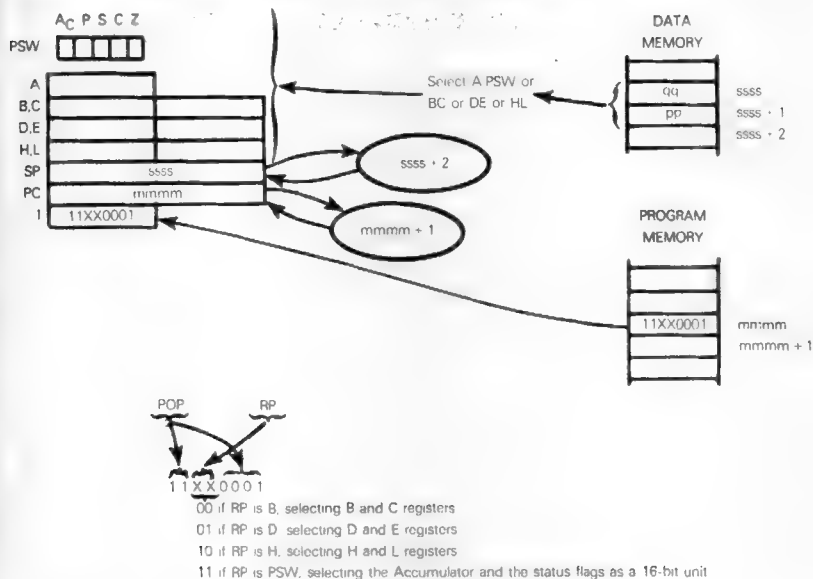
Consider the following call to subroutine SUB:

```
CALL    SUB    ;CALL SUBROUTINE
JMP     ERR    ;ERROR RETURN
          ;GOOD RETURN
```

Using RET to return from SUB would return execution to JMP ERR; therefore, if SUB executes without detecting error conditions, return as follows:

```
POP     H      ;POP RETURN ADDRESS TO HL
INX     H      ;ADD 3 TO RETURN ADDRESS
INX     H
INX     H
PCHL    ;RETURN
```

POP — READ FROM THE TOP OF THE STACK



Pop the two top stack bytes into the designated register pair

Suppose $qq = 03_{16}$ and $pp = 2A_{16}$. Execution of the instruction:

POP R, H

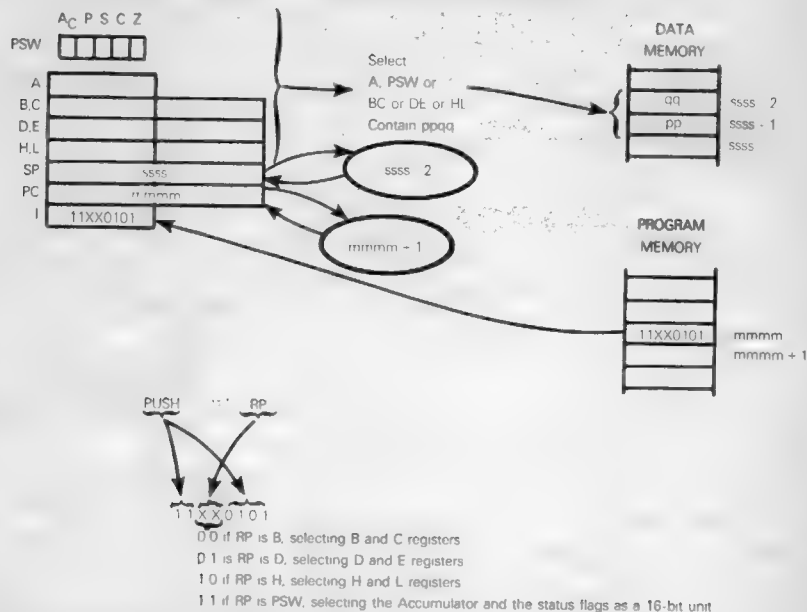
loads 03_{16} into the L register and $2A_{16}$ into the H register. Execution of the instruction:

POP PSW

loads 03_{16} into the status flags and $2A_{16}$ into the Accumulator. Thus the C status will be set to 1; other statuses will be cleared.

The POP instruction is most frequently used to restore register and status contents which have been saved on the stack, for example, while servicing an interrupt.

PUSH — WRITE TO THE TOP OF THE STACK



Push the contents of the designated register pair onto the top of the stack

Suppose the H register contains 03_{16} and the L register contains $2A_{16}$. Execution of the instruction:

PUSH H

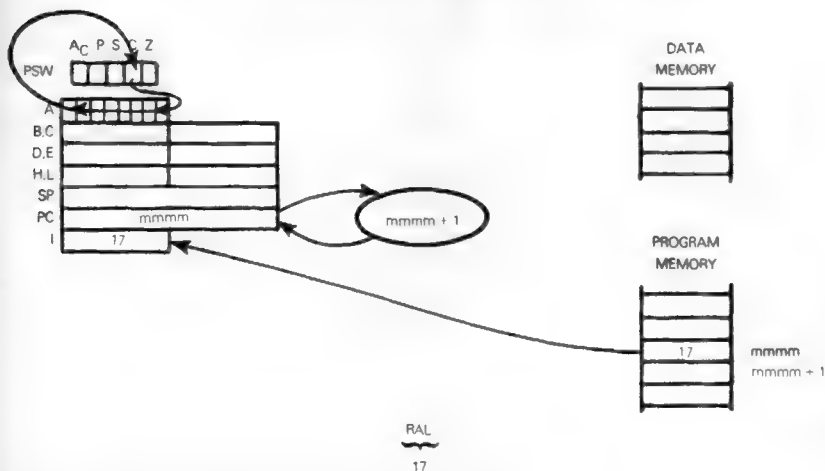
loads 03_{16} then $2A_{16}$ into the top of the stack. Execution of the instruction:

PUSH PSW

loads the Accumulator, then the status flags into the top of the stack.

The PUSH instruction is most frequently used to save register and status contents, for example, before servicing an interrupt.

RAL — ROTATE ACCUMULATOR LEFT THROUGH CARRY



Rotate Accumulator contents left one bit through Carry status.

Suppose the Accumulator contains $7A_{16}$ and the Carry status is set to 1. After the

RAL

instruction is executed, the Accumulator will contain $F5_{16}$ and the Carry status will be reset to 0:

Accumulator	C	→	Accumulator	C
01111010	1		11110101	0

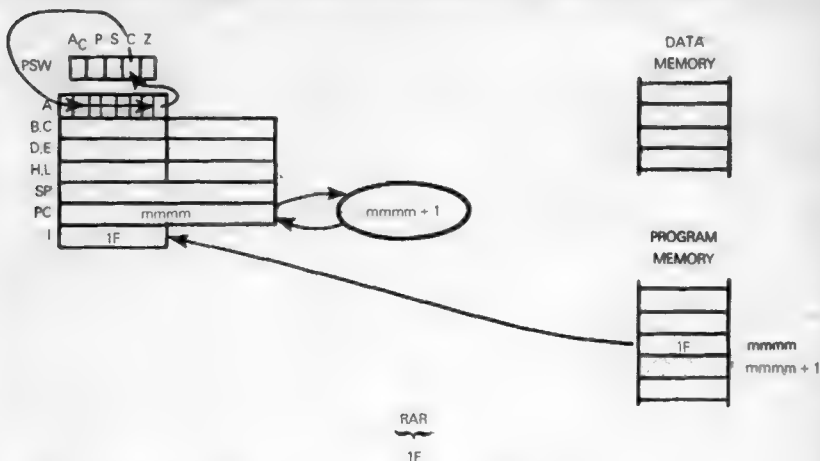
The RAL instruction is frequently used to perform multibyte left shifts, as described in "An Introduction To Microcomputers, Volume I". The Carry status is cleared before performing the first left shift; subsequently the Carry status propagates the high order bit of one byte into the low order bit of the next byte. Here is an instruction sequence that shifts the contents of four memory bytes left, one bit:

LXI	H,DATA	:LOAD ADDRESS OF LOW ORDER DATA BYTE
ANA	A	:INITIALLY CLEAR CARRY
MVI	B,3	:USE REGISTER B AS A COUNTER
LOOP	MOV A,M	:LOAD DATA BYTE INTO ACCUMULATOR
	RAL	:ROTATE LEFT
	MOV M,A	:RESTORE RESULT
	INX H	:INCREMENT ADDRESS IN HL
	DCR B	:DECREMENT COUNTER
	JNZ LOOP	:RETURN FOR NEXT BYTE IF THERE IS ONE

Notice the careful thought that has been given to the statuses that are, or are not set. RAL effects the Carry status only. INX and DCR effect the Zero, Sign and Parity statuses, but not the Carry, which is therefore preserved from one execution of RAL to the next.

**STATUS
CONDITIONS**

RAR — ROTATE ACCUMULATOR RIGHT THROUGH CARRY



Rotate Accumulator contents right one bit through Carry status.

Suppose the Accumulator contains: $7A_{16}$ and the Carry status is set to 1. After the:

RAR

instruction is executed, the Accumulator will contain BD_{16} and the Carry status will be reset to 0:

Accumulator	C	→	Accumulator	C
01111010	1		10111101	0

The RAR instruction is frequently used to perform multibyte right shifts, as described in "An Introduction To Microcomputers, Volume I". The Carry status is cleared before performing the first right shift; subsequently the Carry status propagates the high order bit of one byte into the low order bit of the next byte. Here is an instruction sequence that shifts the contents of four memory bytes right, one bit:

LXI	H, DATA	:LOAD ADDRESS OF LOW ORDER DATA BYTE
ANA	A	:INITIALLY CLEAR CARRY
MVI	B, 3	:USE REGISTER B AS A COUNTER
LOOP	MOV A, M	:LOAD DATA BYTE INTO ACCUMULATOR
	RAR	:ROTATE RIGHT
	MOV M, A	:RESTORE RESULT
	INX H	:INCREMENT ADDRESS IN HL
	DCR B	:DECREMENT COUNTER
	JNZ LOOP	:RETURN FOR NEXT BYTE IF THERE IS ONE

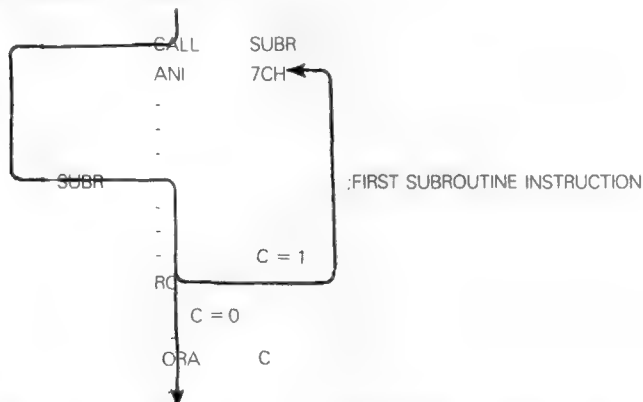
See the RAL description for a discussion of statuses.

RC — RETURN IF THE CARRY STATUS EQUALS 1

RC
D8

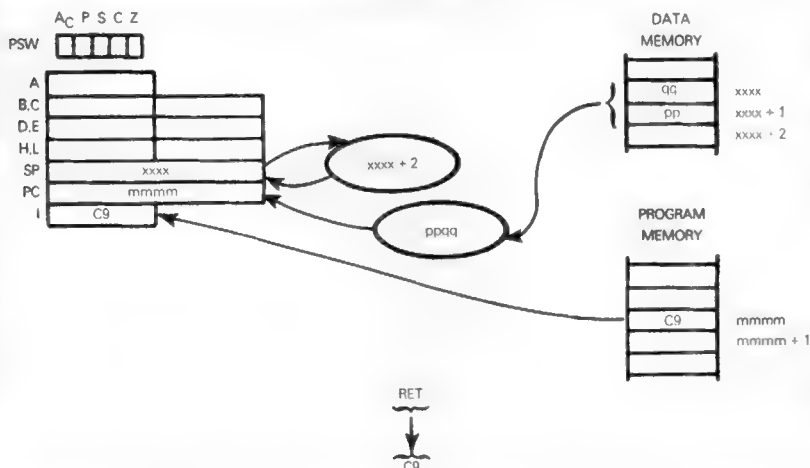
This instruction is identical to the RET instruction except that the return is not executed unless the Carry status equals 1 when the RC instruction is executed.

Consider the instruction sequence:



After the RC instruction is executed, if the Carry status equals 1, execution returns to the ANI instruction which follows the CALL. If the Carry status equals 0, the ORA instruction being the next sequential instruction, is executed.

RET — RETURN FROM SUBROUTINE

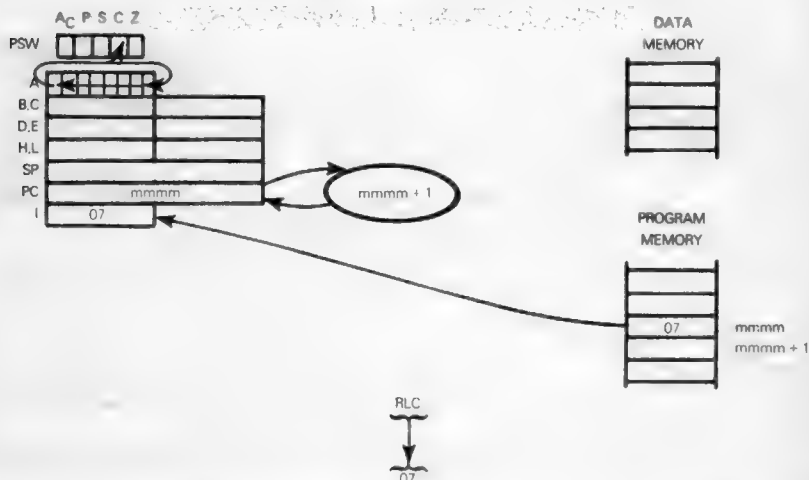


Move the contents of the top two stack bytes to the Program Counter, these two bytes provide the address of the next instruction to be executed. Previous Program Counter contents are lost. Increment the Stack Pointer by 2 to address the new top of stack.

Every subroutine must contain at least one Return (or conditional Return) instruction; this is the last instruction executed within the subroutine and causes execution to return to the calling program.

For an illustrated description of the RET instruction's execution see Chapter 5.

RLC — ROTATE ACCUMULATOR LEFT



Rotate Accumulator contents left one bit.

Suppose the Accumulator contains $7A_{16}$ and the Carry status is set to 1. After the:

RLC

instruction is executed, the Accumulator will contain $F4_{16}$ and the Carry status will be reset to 0:

Accumulator	C	→	Accumulator	C
01111010	1		11110100	0

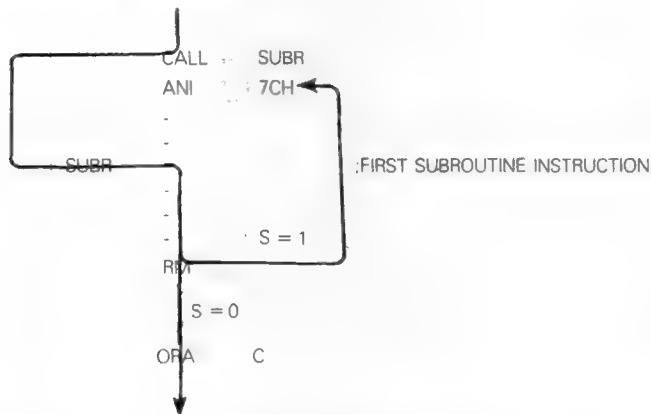
RLC should be used as a logical instruction.

RM — RETURN IF THE SIGN STATUS EQUALS 1

RM
F8

This instruction is identical to the RET instruction except that the return is not executed unless the Sign status equals 1 when the RSM instruction is executed.

Consider the instruction sequence:



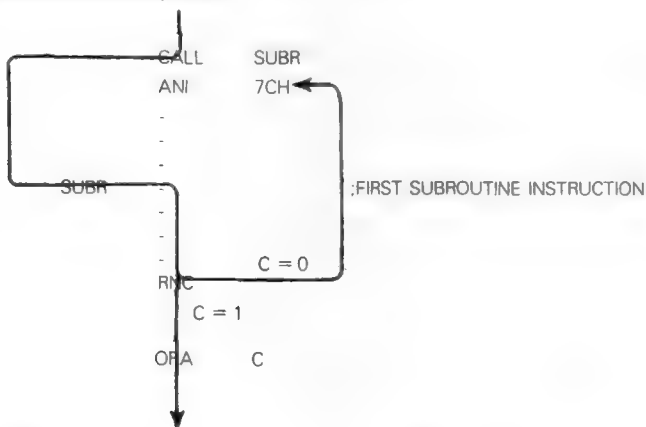
After the RSM instruction is executed, if the Sign status equals 1, execution returns to the ANI instruction which follows the CALL. If the Sign status equals 0, the ORA instruction, being the next sequential instruction, is executed.

RNC — RETURN IF THE CARRY STATUS EQUALS 0

RNC
D0

This instruction is identical to the RET instruction except that the return is not executed unless the Carry status equals 0 when the RNC instruction is executed.

Consider the instruction sequence:



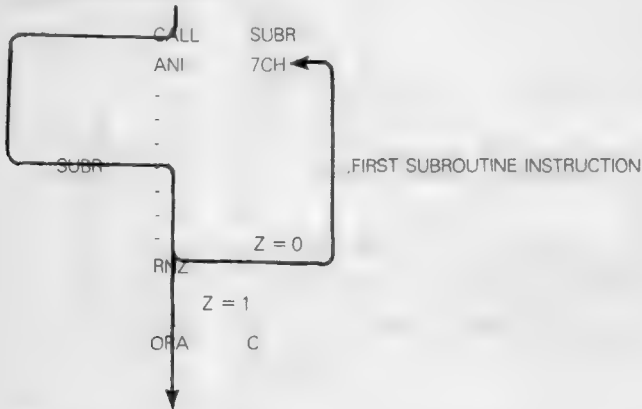
After the RNC instruction is executed, if the Carry status equals 0, execution returns to the ANI instruction which follows the CALL. If the Carry status equals 1, the ORA instruction, being the next sequential instruction, is executed.

RNZ — RETURN IF THE ZERO STATUS EQUALS 0

RNZ
CO

This instruction is identical to the RET instruction except that the return is not executed unless the Zero status equals 0 when the RNZ instruction is executed.

Consider the instruction sequence:



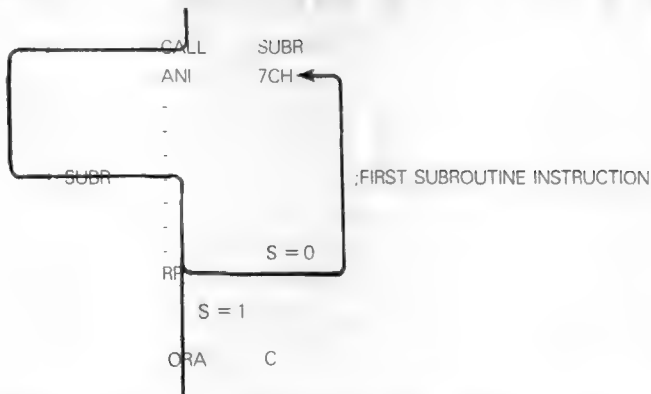
After the RNZ instruction is executed, if the Zero status equals 0, execution returns to the ANI instruction which follows the CALL. If the Zero status equals 1, the ORA instruction, being the next sequential instruction, is executed.

RP — RETURN IF THE SIGN STATUS EQUALS 0

RP
FO

This instruction is identical to the RET instruction except that the return is not executed unless the Sign status equals 0 when the RP instruction is executed.

Consider the instruction sequence:



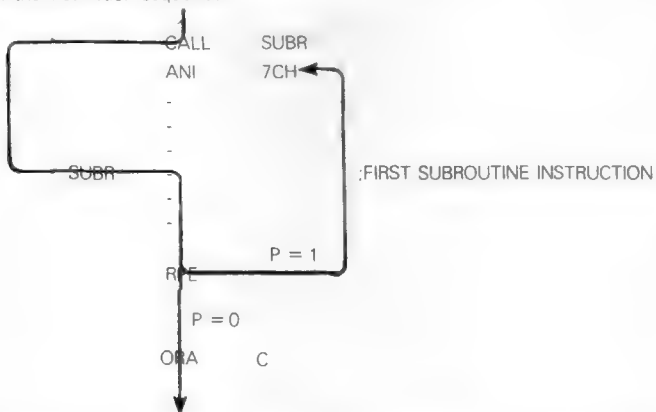
After the RP instruction is executed, if the Sign status equals 0, execution returns to the ANI instruction which follows the CALL. If the Sign status equals 1, the ORA instruction, being the next sequential instruction, is executed.

RPE — RETURN IF THE PARITY STATUS EQUALS 1

RPE
E8

This instruction is identical to the RET instruction except that the return is not executed unless the Parity status equals 1 when the RPE instruction is executed.

Consider the instruction sequence:



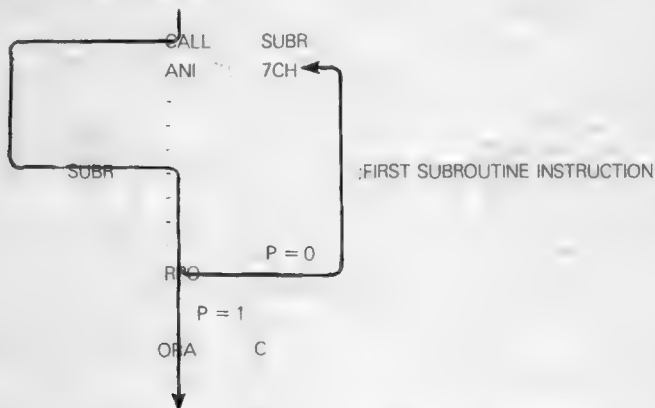
After the RPE instruction is executed, if the Parity status equals 1, execution returns to the ANI instruction which follows the CALL. If the Parity status equals 0, the ORA instruction, being the next sequential instruction, is executed.

RPO — RETURN IF THE PARITY STATUS EQUALS 0

RPO
EO

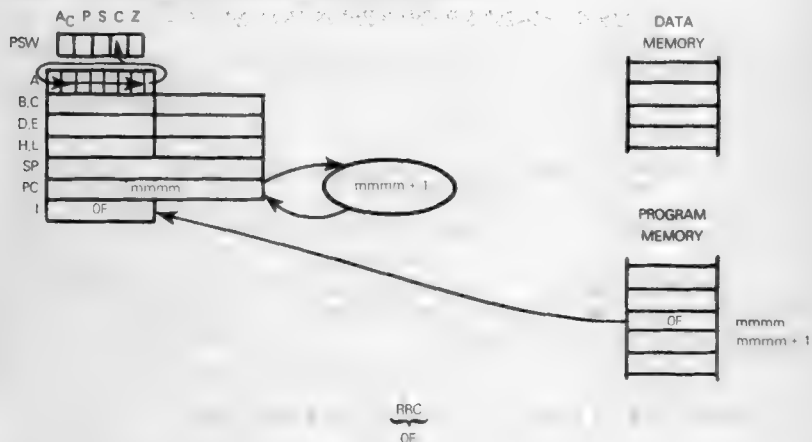
This instruction is identical to the RET instruction except that the return is not executed unless the Parity status equals 0 when the RPO instruction is executed.

Consider the instruction sequence:



After the RPO instruction is executed, if the Parity status equals 0, execution returns to the ANI instruction which follows the CALL. If the Parity status equals 1, the ORA instruction, being the next sequential instruction, is executed.

RRC — ROTATE ACCUMULATOR RIGHT



Rotate Accumulator contents right one bit.

Suppose the Accumulator contains $7A_{16}$ and the Carry status is set to 1. After the

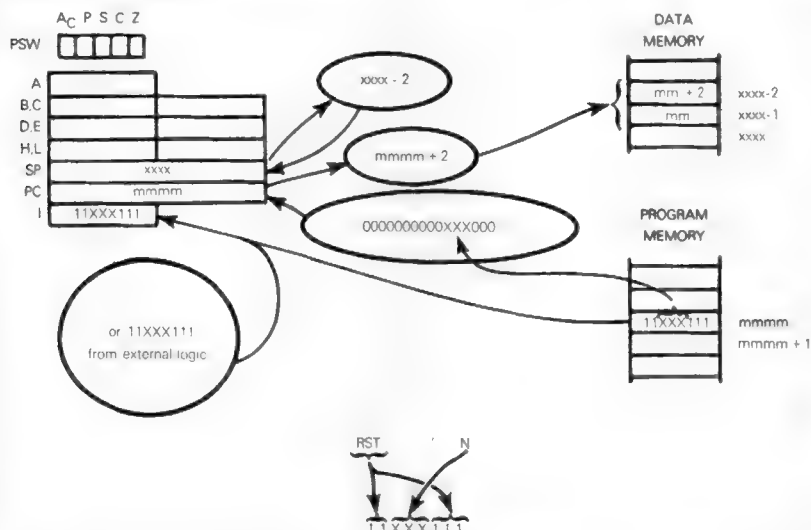
RRC

instruction is executed, the Accumulator will contain $3D_{16}$ and the Carry status will be reset to 0:

Accumulator	C	→	Accumulator	C
01111010	1		00111101	0

RRC should be used as a logical instruction.

RST — RESTART



Call the subroutine originated at the low memory address specified by N.

When the instruction:

RST 3

is executed, the subroutine originated at memory location 0018_{16} is called. The previous Program Counter contents are pushed to the top of the stack.

Usually the RST instruction is used in conjunction with interrupt processing, as described in Chapter 5

If your application does not use all RST instruction codes to service interrupts, do not overlook the possibility of calling subroutines using RST instructions. Origin frequently used subroutines at appropriate RST addresses, and these subroutines can be called with a single byte RST instruction, instead of a three-byte CALL instruction

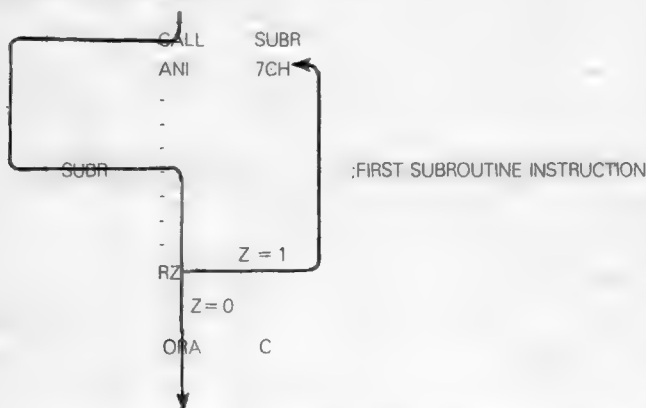
**SUBROUTINE
CALL USING
RST**

RZ — RETURN IF THE ZERO STATUS EQUALS 1

RZ
C8

This instruction is identical to the RET instruction except that the return is not executed unless the Zero status equals 1 when the RZ instruction is executed.

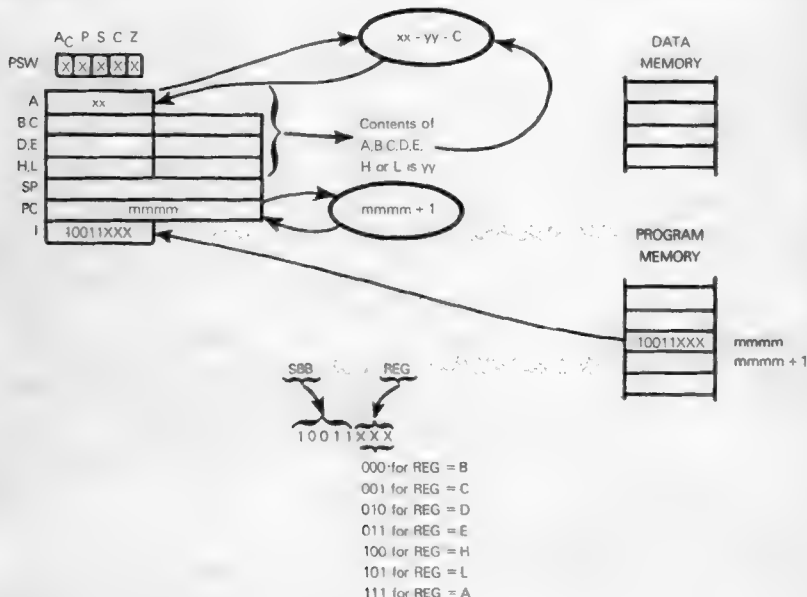
Consider the instruction sequence:



After the RZ instruction is executed, if the Zero status equals 1, execution returns to the ANI instruction which follows the CALL. If the Zero status equals 0, the ORA instruction, being the next sequential instruction, is executed.

SBB — SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR WITH BORROW

This instruction takes two forms. First consider a register's contents subtracted from the Accumulator:

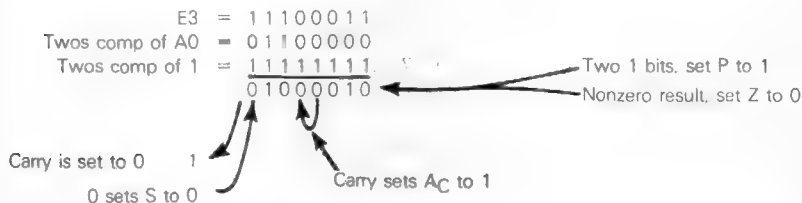


Subtract the contents of the specified register, and the Carry status, from the Accumulator treating registers contents as simple binary data.

Suppose $xx = E3_{16}$, register E contains $A0_{16}$, $C = 1$. After instruction:

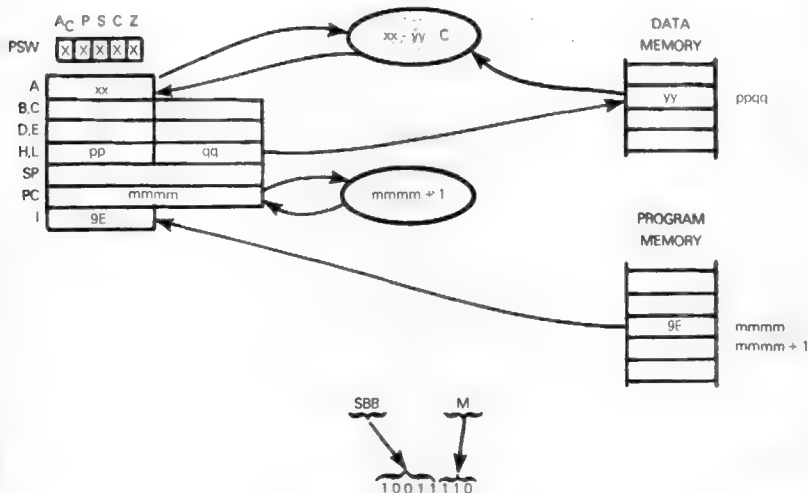
SBB E

has executed, the Accumulator will contain 42_{16} :



Notice that the resulting Carry is complemented.

The contents of a memory byte may also be subtracted, with borrow, from the Accumulator:



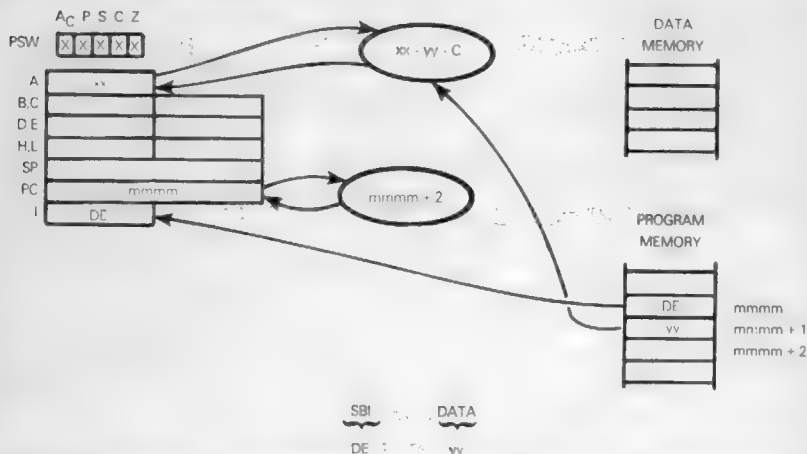
If $xx = E3_{16}$ and $yy = A0_{16}$, $C = 1$ then execution of the instruction:

SBB M

generates the same result as execution of the SBB E instruction, which was just described.

The SBB instruction is used in multibyte subtraction after the low order byte has been processed using the SUB instruction.

SBI — SUBTRACT IMMEDIATE DATA FROM ACCUMULATOR WITH BORROW



Subtract the contents of the second instruction code byte, and the Carry status, from the Accumulator.

Suppose $xx = 3A_{16}$ and the Carry status = 1. After the instruction:

SBI 7CH

has executed, the Accumulator will contain BD_{16} :

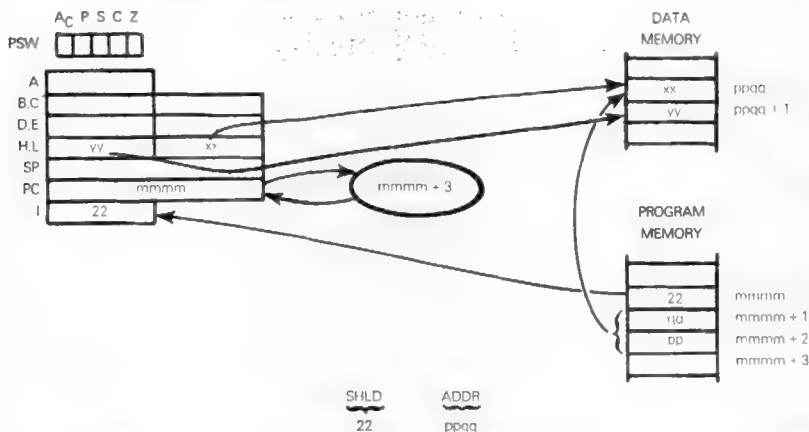
3A	=	00111010	
Twos comp of 7C	=	10000100	
Twos comp of Carry	=	11111111	
		<u>10111101</u>	← Six 1 bits, set P to 1
			← Nonzero result, set Z to 0

Carry is set to 0 — 1
1 sets S to 1 —
Carry sets A_C to 1

Notice that the resulting Carry is complemented.

This instruction is not used as frequently as SUI.

SHLD — STORE H AND L REGISTERS DIRECT



The second and third object code bytes provide the memory address of a data byte into which the L register contents is written. The H register contents are written into the next sequential data byte.

Suppose $xx = 2C_{16}$ and $yy = 3A_{16}$. After the instruction

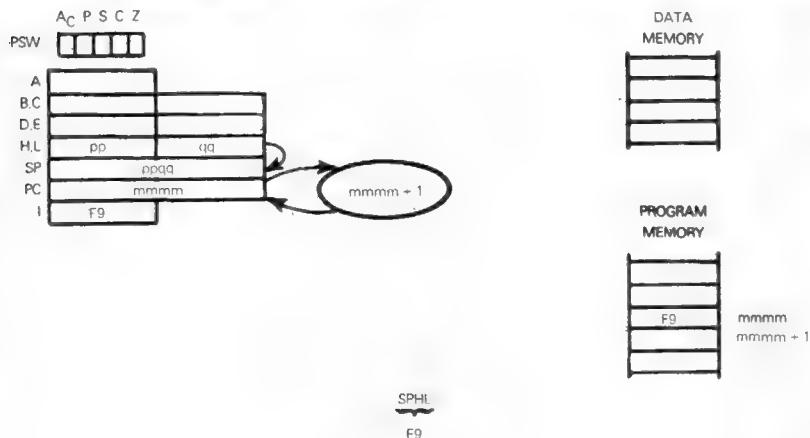
```

LABEL EQU 084AH
      .
      .
      .
      SHLD LABEL
    
```

has executed, memory byte $084A_{16}$ will contain $2C_{16}$. Memory byte $084B_{16}$ will contain $3A_{16}$.

Remember, EQU is an Assembler Directive, it is not an instruction; it tells the Assembler to use the 16-bit value $084A_{16}$ whenever LABEL appears.

SPHL — LOAD THE STACK POINTER FROM THE H AND L REGISTERS



Move the contents of the H and L registers to the Stack Pointer.

Suppose $pp = 08_{16}$ and $qq = 3F_{16}$. After the instruction:

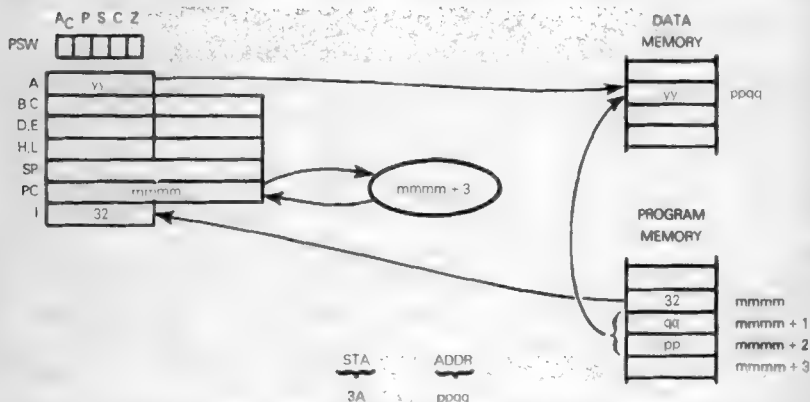
SPHL

has executed, the Stack Pointer will contain $083F_{16}$.

The SPHL instruction can be used to access two stacks — with an idle address preserved in the H and L registers. Frequently stacks are used in this fashion to access text strings or any such data that must be accessed byte serially.

The important point to bear in mind is that stack logic can be used instead of implied memory addressing with auto-increment.

STA — STORE ACCUMULATOR IN MEMORY USING DIRECT ADDRESSING



Store the Accumulator contents in the memory byte addressed directly by the second and third bytes of the STA instruction object code

Suppose the Accumulator contains $3A_{16}$. After the instruction:

`LABEL EQU 084AH`

`STA LABEL`

has executed, memory byte $084A_{16}$ will contain $3A_{16}$.

Remember, EQU is an Assembler Directive, it is not an instruction; it tells the Assembler to use the 16-bit value $084A_{16}$ wherever LABEL appears

The instruction:

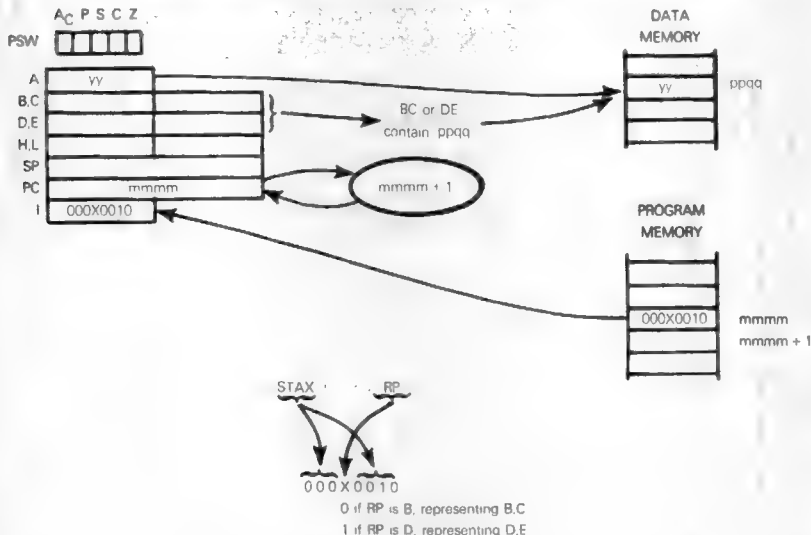
`STA LABEL`

is equivalent to the two instructions:

`LXI H, LABEL`
`MOV M, A`

When you are storing a single data value in memory the STA instruction is preferred. It uses one instruction and three object program bytes to do what the LXI MOV combination does in two instructions and four object program bytes. Also, the LXI MOV combination uses the H and L registers; the STA instruction does not.

STAX — STORE ACCUMULATOR IN THE MEMORY LOCATION ADDRESSED BY A REGISTER PAIR



Store the Accumulator contents in the memory byte addressed by the BC, or DE register pair.

Suppose the B register contains 08_{16} , the C register contains $4A_{16}$ and the Accumulator contains $3A_{16}$. After the instruction:

STAX B

has executed, memory byte $084A_{16}$ will contain $3A_{16}$.

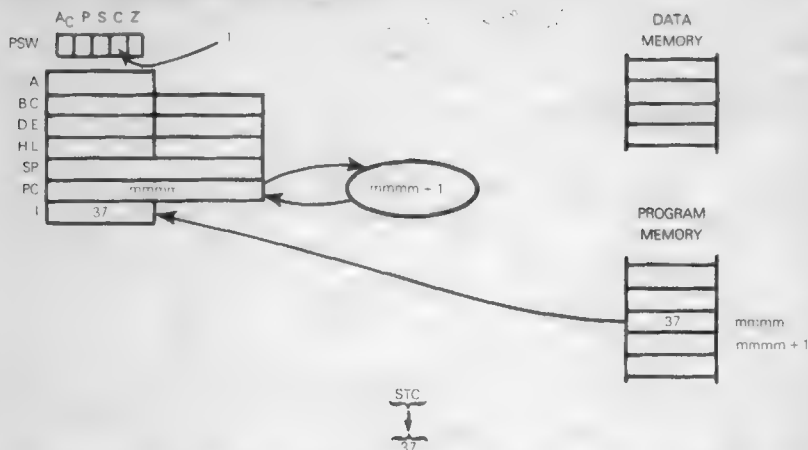
Note that there is no STAX H instruction since this is identical to a MOV M,A instruction.

Normally the STAX and LXI instructions will be used together, since the LXI instruction loads a 16-bit address into the BC or DE registers, as follows:

```
LXI    B,084AH
STAX   B
```

Notice that the STAX instruction will only store data from the Accumulator, whereas the MOV instruction will store data from any register.

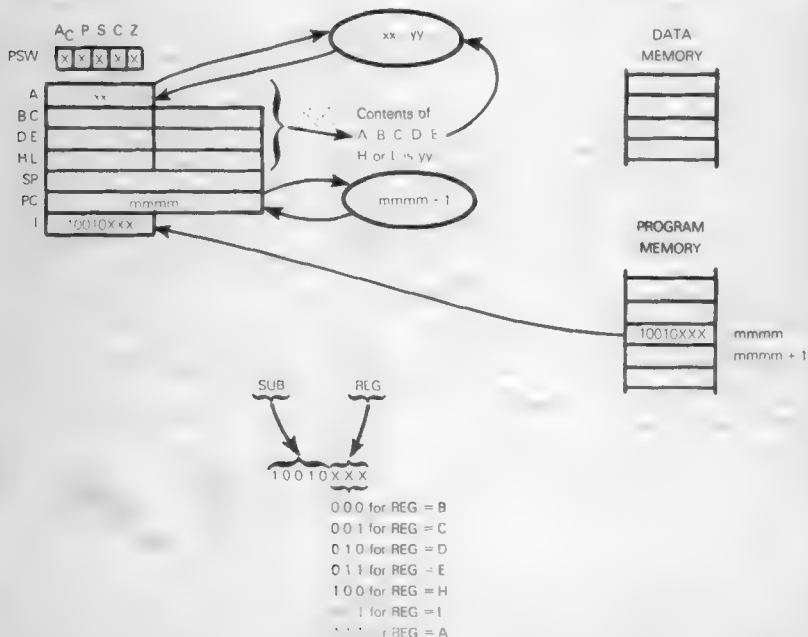
STC — SET CARRY STATUS



When the STC instruction is executed, the Carry status is set to 1, regardless of its previous value. No other statuses or registers contents are effected.

SUB — SUBTRACT REGISTER OR MEMORY FROM ACCUMULATOR

This instruction takes two forms. First consider a register's contents subtracted from the Accumulator:



Subtract the contents of the specified register from the Accumulator treating registers contents as simple binary data.

Suppose $xx = E3_{16}$ and register E contains $A0_{16}$. After instruction:

SUB E

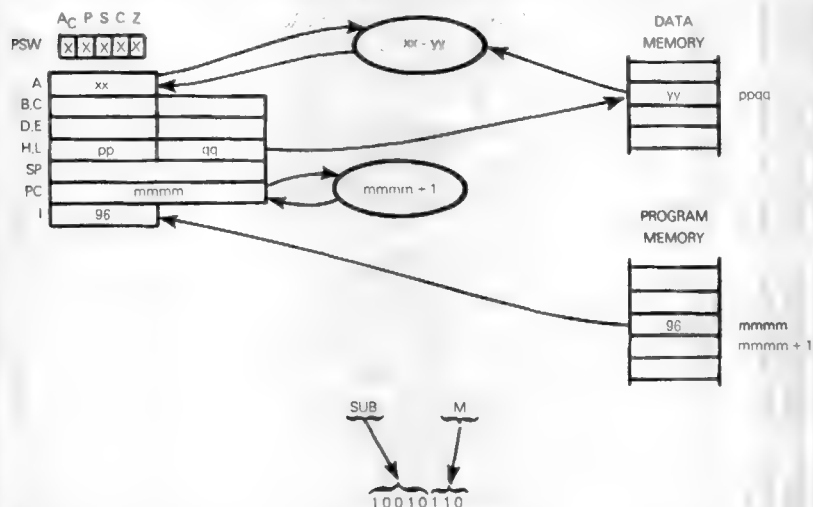
has executed, the Accumulator will contain 43_{16} :

$$\begin{array}{r}
 E3 = 11100011 \\
 \text{Twos comp of } A0 = 01100000 \\
 \hline
 01000011
 \end{array}$$

Three 1 bits, set P to 0
 Nonzero result, set Z to 0
 Carry is set to 0-1
 No carry so A_C is set to 0
 0 sets S to 0

Notice that the resulting Carry is complemented.

The contents of a memory byte may also be subtracted from the Accumulator:



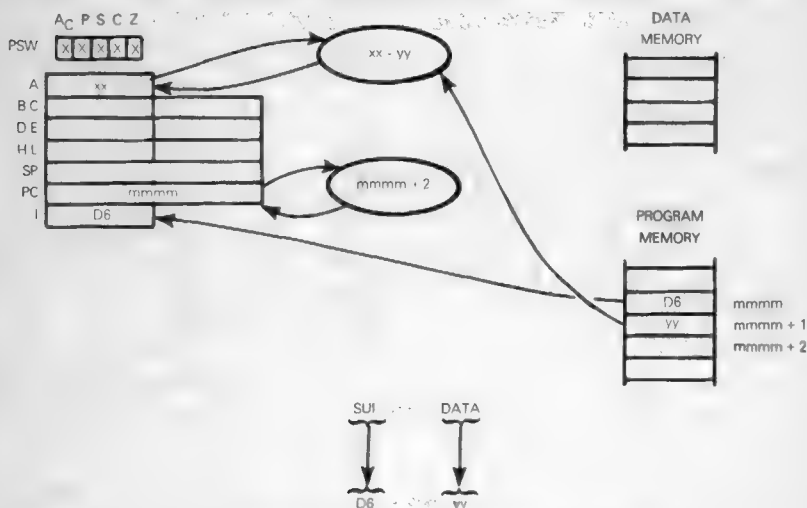
If $xx = E3_{16}$ and $yy = A0_{16}$, then execution of the instruction:

SUB M

generates the same result as execution of the **SUB E** instruction, which was just described.

The **SUB** instruction is used to perform single byte subtractions, or for the low order byte in multibyte subtractions.

SUI — SUBTRACT IMMEDIATE DATA FROM ACCUMULATOR



Subtract the contents of the second instruction code byte from the Accumulator.

Suppose $xx = 3A_{16}$. After the instruction:

SBI 7CH

has executed, the Accumulator will contain BE_{16} :

$$\begin{array}{r}
 3A_{16} = 00111010 \\
 \text{Twos comp of } 7C_{16} = 10000100 \\
 \hline
 10111110
 \end{array}$$

Six 1 bits, set P to 1

Nonzero result, set Z to 0

No Carry so Carry is set to 1

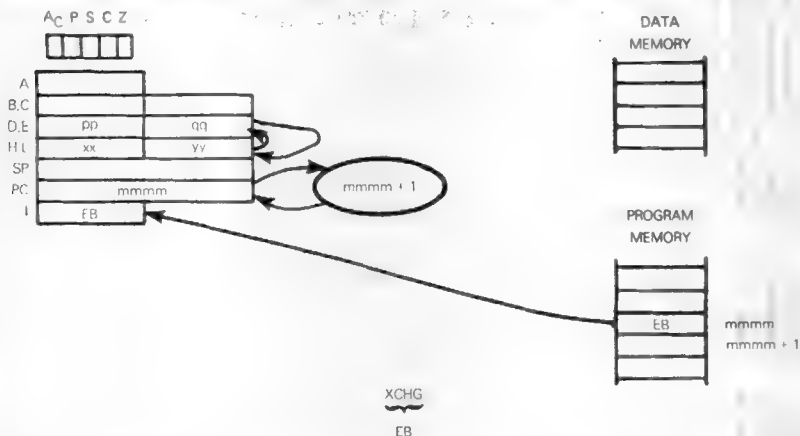
-1 sets S to 1

No Carry so A_C is set to 0

Notice that the resulting Carry is complemented.

This instruction is the preferred subtract immediate.

XCHG — EXCHANGE DE AND HL REGISTERS' CONTENTS



The D and E registers' contents are swapped with the H and L registers' contents

Suppose $pp = 03_{16}$, $qq = 2A_{16}$, $xx = 41_{16}$ and $yy = FC_{16}$. After the instruction:

XCHG

has executed, H will contain 03_{16} , L will contain $2A_{16}$, D will contain 41_{16} and E will contain FC_{16}

The two instructions:

```
XCHG
MOV   A,M
```

are equivalent to:

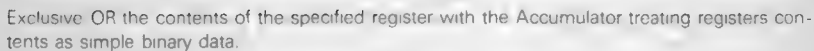
```
LDAX  D
```

But if you want to load data addressed by the D and E registers into the B register,

```
XCHG
MOV   B,M
```

has no single instruction equivalent.

This instruction takes two forms. First consider a register's contents Exclusive-ORed with the Accumulator:

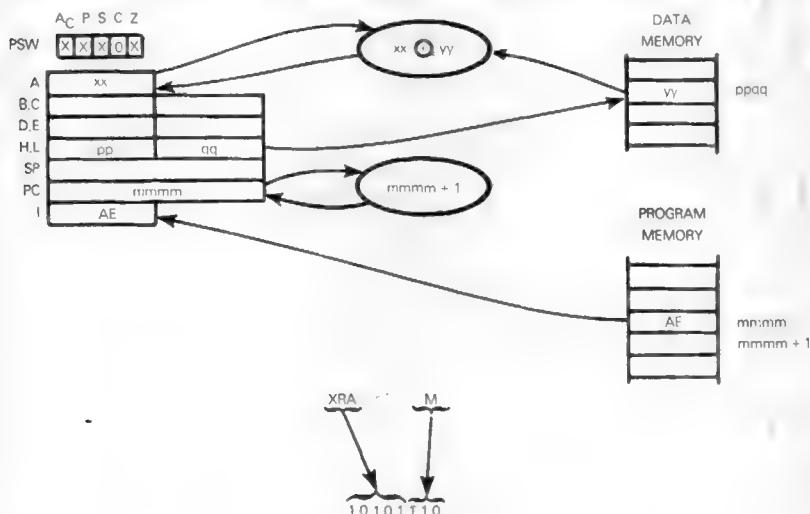


XRA E

has executed, the Accumulator will contain 43₁₆:

6-76

The contents of a memory byte may also be Exclusive-ORed with the Accumulator



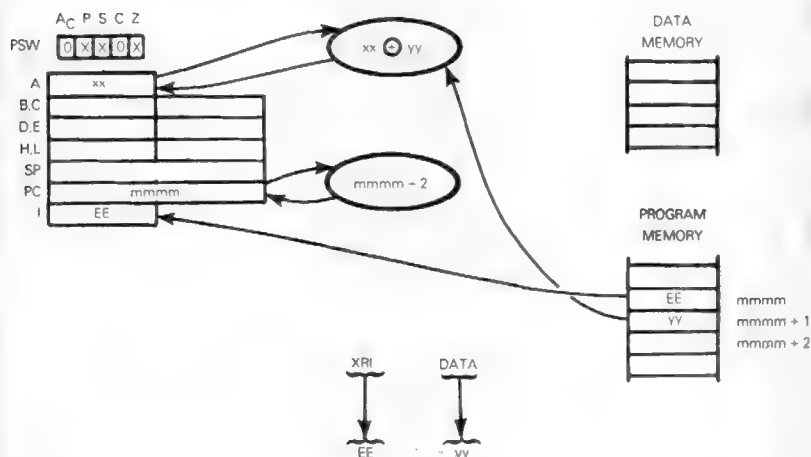
If $xx = E3_{16}$ and $yy = A0_{16}$, then execution of the instruction:

XRA M

generates the same result as execution of the XRA E instruction, which was just described.

The Exclusive-OR instruction is used to test for changes in bit status.

XRI — EXCLUSIVE-OR IMMEDIATE DATA WITH ACCUMULATOR



Exclusive-OR the contents of the second instruction code byte with the Accumulator

Suppose $xx = 3A_{16}$. After the instruction:

XRI 7CH

has executed, the Accumulator will contain 46_{16} :

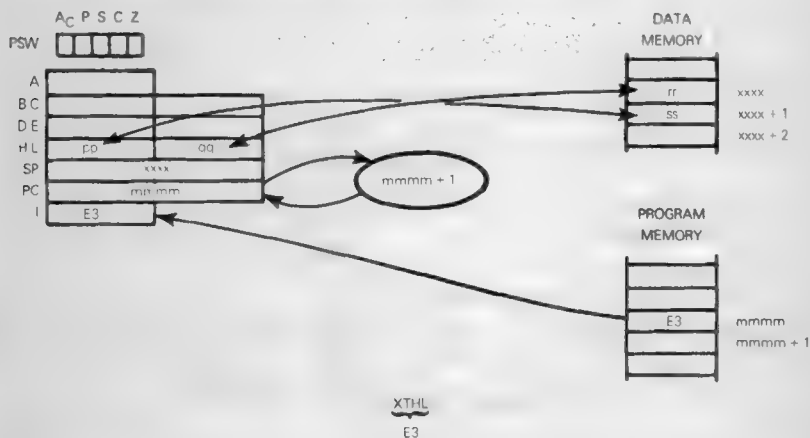
3A =	00111010	
Twos comp of 7C =	01111100	Three 1 bits, set P to 1
	01000110	Nonzero result, set Z to 0

Carry is set to 0

0 sets S to 0

This instruction is the preferred subtract immediate.

XTHL — EXCHANGE TOP OF STACK WITH HL



Exchange the contents of the L register with the top stack byte. Exchange the contents of the H register with the byte below the stack top.

Suppose $pp = 21_{16}$, $qq = FA_{16}$, $rr = 3A_{16}$, $ss = E2_{16}$. After the instruction:

XTHL

is executed, H will contain $E2_{16}$, L will contain $3A_{16}$ and the top two stack bytes will contain FA_{16} and 21_{16} , respectively.

The two instructions:

XTHL
XTHL

executed in sequence are equivalent to no operation.

The XTHL instruction is used to access and manipulate data at the top of the stack, as illustrated in the multiple subroutine discussion in Chapter 5.

Chapter 7

SOME COMMONLY USED SUBROUTINES

There are a number of operations which occur in many microcomputer programs, irrespective of the application. This chapter will provide a number of frequently used instruction sequences.

To make the most effective use of this chapter, you should study each subroutine until you know it well enough to modify it. As a simple exercise, you should attempt to rewrite the subroutine, so that it does the same job using fewer execution cycles, or fewer instructions, or both. Next, rewrite the programs to implement variations. For example, binary multiplication of 16 bit numbers is illustrated, how about a routine to multiply 32-bit numbers? Look upon each example as a typical, illustrative instruction sequence, which you will likely modify to meet your immediate needs.

Simple programs at the level covered in this chapter fall into one of four categories:

- 1) Memory addressing**
- 2) Data movement**
- 3) Arithmetic**
- 4) Program execution sequence logic**

We will describe programs in the above category sequence.

MEMORY ADDRESSING

Although the 8080/9080 memory addressing modes are limited to direct and implied addressing, simple instruction sequences allow any of the other addressing modes to be simulated.

We are going to show auto increment and auto decrement, indexed addressing, indirect addressing and indirect addressing with post indexing, all of these addressing modes are described and illustrated in "An Introduction To Microcomputers", Volume I

AUTO INCREMENT AND AUTO DECREMENT

One of the weaknesses of the 8080/9080 instruction set, as compared to some other microcomputers, is the lack of auto incrementing and auto decrementing implied addressing options; the data move routines described later in this chapter illustrate the gratuitous need to constantly increment/decrement addresses when handling data buffers — or any blocks of contiguous data memory bytes

Under some circumstances **you can use the Stack Pointer to implement implied memory addressing with auto increment or auto decrement. However, you must live with some programming restrictions:**

STACK POINTER MEMORY ADDRESSING
--

- 1) You must use the Push instruction in lieu of a write-to-memory and the Pop instruction in**

lieu of a read from memory. This restricts you to auto decrementing when writing and auto incrementing when reading.

- 2) Memory is accessed as byte pairs; remember, the Push and Pop instructions deal with register pairs, not with single-byte data units. This can be an advantage if it cuts in half the number of instructions executed; it is a disadvantage if you are dealing with buffers of odd-byte length, or if for any reason you cannot handle your data in 16-bit units.
- 3) The previous Stack Pointer contents address the current stack top, so it must be saved while using the Stack Pointer as a memory address register. This, of course, means you cannot use subroutines, or access the stack, until you have restored the Stack Pointer.

How are you going to save the prior Stack Pointer contents? The 8080 instruction set provides the SPHL instruction to load the Stack Pointer from the H and L registers; but **there is no instruction which moves Stack Pointer contents into the HL registers — or any other**

register. Instead, you must clear the H and L registers, then execute a DAD instruction as follows:

```
MVI 00H,00H :CLEAR H
MOV 00H,00H :CLEAR L
DAD SP,SP :MOVE SP TO HL
```

Note that the DAD instruction will modify the Carry status. Now you can save the Stack Pointer contents in another register pair:

```
XCHG 00H,00H :SAVE HL IN DE
```

or

```
MOV 00H,00H :SAVE HL IN BC
MOV 00H,00H :C,L
```

or you can reserve two bytes of read-write memory for temporary Stack Pointer storage

```
SHLD 00H,00H :Save HL at STAK and STAK + 1
```

Restoring the saved Stack Pointer contents is very easy, from BC, these instructions apply:

```
MOV 00H,00H :MOVE BC TO HL
MOV 00H,00H :L,C
SPHL 00H,00H :MOVE HL TO SP
```

From DE, these instructions apply:

```
XCHG 00H,00H :MOVE DE TO HL
SPHL 00H,00H :MOVE HL TO SP
```

From memory, these instructions apply:

```
LHLD 00H,00H :LOAD HL FROM STAK AND STAK + 1
SPHL 00H,00H :MOVE HL TO SP
```

Once the Stack Pointer contents have been saved, you can load a new address into the Stack Pointer immediately:

```
LXI 00H,00H :SP,ADDR
```

Or you can load an address which has been saved in two bytes of read/write memory:

```
LHLD 00H,00H :LOAD ADDRESS INTO HL
SPHL 00H,00H :MOVE HL TO SP
```

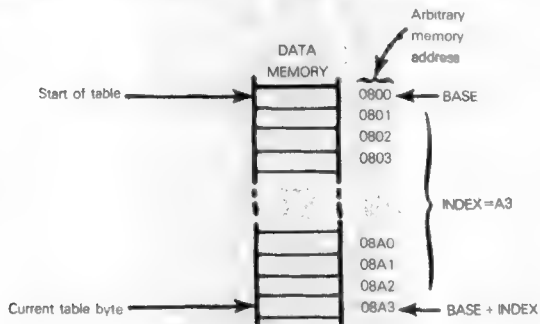
**SAVING
STACK
POINTER**

**RESTORING
STACK
POINTER**

**LOADING
ADDRESS
INTO STACK
POINTER**

INDEXED ADDRESSING

Indexed addressing requires an address to be computed as a base address, plus displacement, provided by an index register; here is an example:



Chances are that BASE is a non-varying address, whereas INDEX is constantly changing; we will therefore access BASE as an immediate, 16 bit value, but INDEX is assigned two read/write memory bytes, with addresses $INDX$ and $INDX + 1$. Now you can create an indexed address as follows:

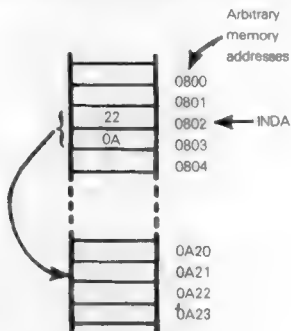
```
LXI    B,BASE    ;LOAD BASE ADDRESS INTO BC
LHLD   INDX      ;LOAD INDEX INTO HL
DAD    B          ;ADD BC TO HL
```

What if the current index value changes? When you are done, you can restore the new index value by subtracting BASE from the contents of HL, as follows:

```
LXI    B,BASE    ;LOAD BASE ADDRESS INTO BC
MOV    A,L        ;SUBTRACT C FROM L
SUB    C           ;SUBTRACT C FROM L
STA    INDX       ;SAVE IN INDX
MOV    A,H        ;SUBTRACT B FROM H WITH BORROW
SBB    B           ;SUBTRACT B FROM H WITH BORROW
STA    INDX + 1   ;SAVE IN INDX + 1
```

INDIRECT ADDRESSING

Indirect addressing specifies that the memory address you require is stored in two memory bytes



In the illustration above, memory bytes 0802₁₆ and 0803₁₆ hold the required memory address 0A22₁₆. In keeping with the way the 8080 itself handles 16-bit addresses, the low order address byte is shown preceding the high order address byte.

These instructions simulate indirect addressing:

```
LHLD    INDA    ;LOAD ADDRESS INTO HL
NOW ACCESS MEMORY USING NORMAL, IMPLIED
ADDRESSING, WITH HL PROVIDING THE ADDRESS
```

INDIRECT, POST-INDEXED ADDRESSING

Look again at the table we accessed using indexed addressing. Suppose the table base address (BASE) is found in two memory bytes, addressed by INDA and INDA + 1. You can complete the current table address as follows:

LHLD	INDA	;LOAD BASE ADDRESS INTO HL
MOV	B,H	;SAVE IN B,C
MOV	C,L	
LHLD	INDX	;LOAD INDEX INTO HL
DAD	B	;ADD BC TO HL

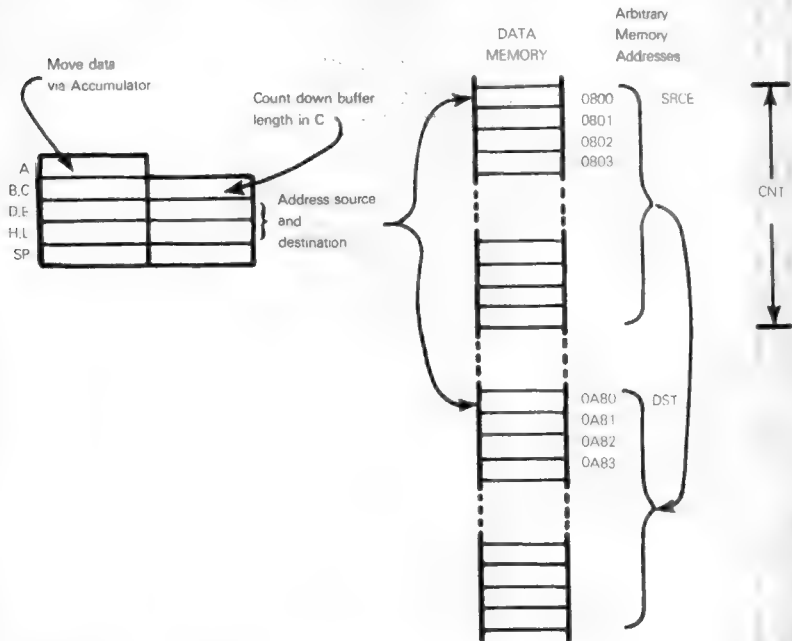
This is equivalent to indirect, post-indexed addressing.

DATA MOVEMENT

We will now examine some instruction sequences that locate and move contiguous blocks of data bytes — data buffers of any length.

MOVING SIMPLE DATA BLOCKS

Beginning with a very simple program, consider moving the contents of a contiguous block of data memory bytes from one area of memory to another. The simplest way of performing this operation is to address the source and destination buffers using the DE and HL registers. The following memory map illustrates the data movement operation:



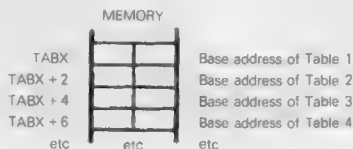
This is the data move program:

	LXI	H,SRCE	:LOAD SOURCE ADDRESS INTO HL
	LXI	D,DST	:LOAD DESTINATION ADDRESS INTO DE
	MVI	C,CNT	:LOAD BYTE COUNT INTO C
LOOP	MOV	A,M	:LOAD SOURCE BYTE
	INX	H	:INCREMENT SOURCE ADDRESS
	STAX	D	:STORE IN DESTINATION
	INX	D	:INCREMENT DESTINATION ADDRESS
	DCR	C	:DECREMENT BUFFER LENGTH
	JNZ	LOOP	:RETURN IF BUFFER NOT EMPTY

MULTIPLE TABLE LOOKUPS

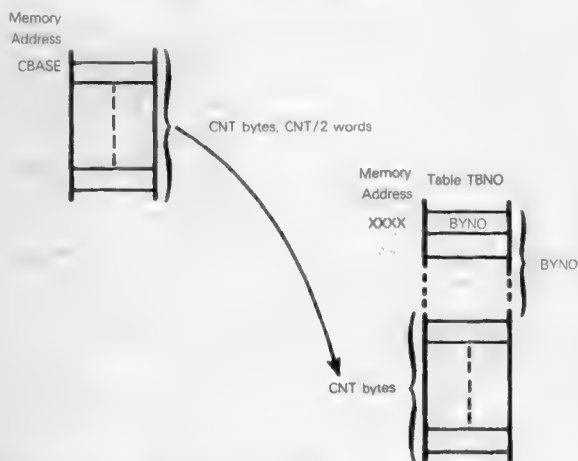
Next consider a multiple table lookup. This is a more complex variation of the data move which we just described.

An indefinite number of data tables have their starting addresses stored in an index table. The index table's starting address is given by label TABX:



A number of data bytes are in temporary storage, starting at a memory location identified by the label CBASE. The actual number of data bytes can be found in a memory location identified by the label CNT. This source buffer is equivalent to the source buffer in the data move program we have just described.

The destination for the block of data is one of the data tables. The table number is identified by the symbol TBNO, which is loaded as immediate data. The first two bytes of every table identify the displacement to the first free byte of the table, in other words, we assume that every table is partially filled and the block of data is to be moved into the unoccupied end of the selected table. The required data movement may be illustrated as follows:



Here is the appropriate instruction sequence:

LXI	D,TABX	:LOAD BASE ADDRESS OF TABLE INDEX
LXI	H,TBNO	:LOAD TABLE NUMBER INTO HL
DAD	D	:COMPUTE ADDRESS OF TABLE BASE ADDRESS
MOV	E,M	:LOAD TABLE BASE ADDRESS INTO DE
INX	H	
MOV	D,M	
XCHG		:MOVE ADDRESS TO HL
MOV	E,M	:LOAD DISPLACEMENT TO FIRST FREE BYTE INTO DE
INX	H	
MOV	D,M	
DAD	D	:ADD TO HL, GIVING ADDRESS OF FIRST FREE BYTE
LXI	D,CBASE	:LOAD INPUT BUFFER BASE ADDRESS INTO DE

	LDA	CNT	:LOAD BYTE COUNTER AND SAVE IN B
	MOV	B,A	
LOOP	LDAX	D	:MOVE NEXT BYTE FROM MEMORY LOCATION
	MOV	M,A	:ADDRESSED BY DE TO LOCATION ADDRESSED BY HL
	INX	D	:INCREMENT SOURCE AND DESTINATION
	INX	H	:ADDRESSES
	DCR	B	:DECREMENT BYTE COUNTER
	JNZ	LOOP	:RETURN FOR MORE BYTES

SORTING DATA

Both of the programming examples we have described thus far simply move a block of data from one location to another. Reorganizing data is also very important, therefore **we will illustrate a sort routine.**

The sort, as illustrated, takes a sequence of signed binary numbers, stored in contiguous memory locations and reorganizes them in ascending order, so that the smallest number comes first and the largest number comes last.

The sort routine we are going to program uses a bubble-up algorithm. Consider a sequence of numbers, where the label LIST identifies the address of the first number's storage location in memory. These are the necessary sort routine program steps:

- 1) Start a pass at the beginning of the LIST, initialize a flag to indicate a "no swap" condition.
- 2) Compare a consecutive pair of numbers; if the first number is smaller than the second number, do nothing; otherwise exchange the two numbers and set the flag to indicate "swap made".
- 3) Compare the address of the second number to the end of list address, identified by the label ENDL. If not at the end, increment so that the second number of the current pair becomes the first number of the next pair and return to step 2.
- 4) At the end of the list, check the "swap" flag. If any swap was made during the pass, return to step 1 to make another pass.
- 5) If a pass is made with no swaps, all numbers are in order. Exit.

**SORTING
DATA**

As an example, consider the case where the numbers 1 through 10 are in reverse order. Nine exchanges will be made during the first pass, at the end of which the largest number will have been "bubbled up" to the top:

	START	AFTER 1 PASS
LIST	10	9
	9	8
	8	7
	7	6
	6	5
	5	4
	4	3
	3	2
	2	1
ENDL	1	10

Another eight passes will be needed to get all numbers in order, then a tenth pass is needed to get a "no swap" exit condition.

SORT is implemented as a subroutine which is passed parameters in locations following the subroutine call. Two parameters are specified.

LIST the beginning address of the data buffer containing numbers to be sorted.
 ENDL the ending address of the data buffer containing numbers to be sorted.

Here is the sort program:

	CALL	SORT	:CALLING SEQUENCE
	DW	LIST	:ADDRESS OF START OF LIST
	DW	ENDL	:ADDRESS OF END OF LIST (ON SAME PAGE AS LIST)
SORT	POP	H	:UNSTACK RETURN ADDRESS INTO HL
	MOV	E,M	:LOAD LIST ADDRESS TO DE
	INX	H	
	MOV	D,M	
	INX	H	
	MOV	C,M	:LOAD LO BYTE OF ENDL ADDRESS TO C
	INX	H	
	INX	H	:INCREMENT PAST HO BYTE OF ENDL ADDRESS
	PUSH	H	:STACK RETURN
	MOV	H,D	:MOVE HO BYTE OF LIST ADDRESS FROM D TO H
	MVI	D,0	:ZERO D AS A "NO SWAP" INDICATOR
LOOP1	MOV	L,E	:MOVE LO BYTE OF LIST ADDRESS FROM E TO L
LOOP2	MOV	A,M	:LOAD 1ST NUMBER OF PAIR TO A
	INR	L	:INCREMENT LIST POINTER
	CMP	M	:COMPARE (SUBTRACT MEMORY FROM A)
	JM	SORT1	:JUMP IF MINUS (2ND NUMBER ALREADY > 1ST NUMBER)
	MOV	B,A	:MOVE 1ST NUMBER TO B FROM A
	MOV	A,M	:LOAD 2ND NUMBER TO A
	DCR	L	:DECREMENT TO 1ST NUMBER LOCATION
	MOV	M,A	:STORE 2ND NUMBER FROM A
	INR	L	:INCREMENT LIST POINTER
	MOV	M,B	:STORE 1ST NUMBER FROM B
	MVI	D,1	:LOAD D WITH CONSTANT 1 AS A "SWAP MADE" FLAG
SORT1	MOV	A,C	:MOVE LO BYTE OF ENDL ADDRESS TO A
	CMP	L	:COMPARE WITH LO BYTE OF LIST ADDRESS IN L
	JNZ	LOOP2	:LOOP BACK IF NOT AT END BYTE
	DCR	D	:DECREMENT FLAG IN D
	JZ	LOOP1	:LOOP BACK TO START LIST AGAIN IF SWAP WAS MADE
	RET		:RETURN

ARITHMETIC

Addition, subtraction, multiplication and division will be described under this group. Transcendental functions are complex enough to require entire text books devoted to the subject, so we will not even broach the subject.

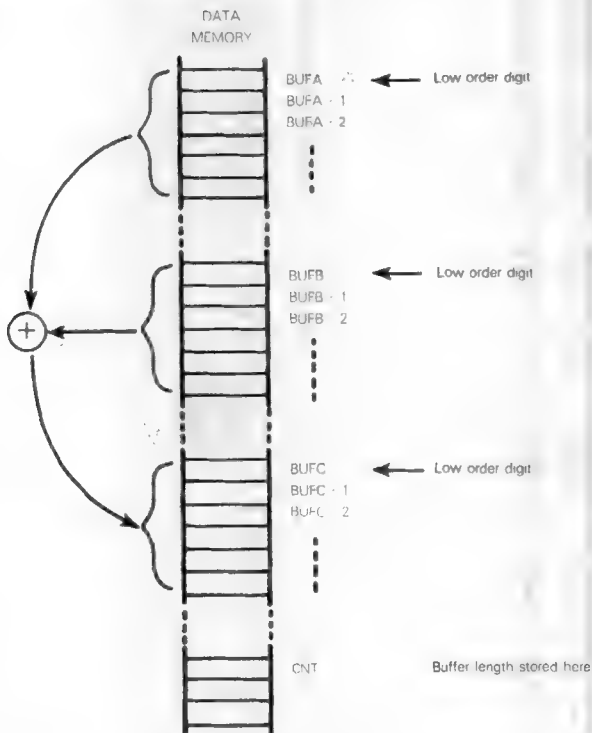
Even within the simple bounds of addition, subtraction, multiplication and division, there is a degree of latitude that exceeds the scope of material we can cover. Significantly different algorithms are required depending upon the magnitude of the number. Binary and decimal arithmetic also require different algorithms. Therefore, **for addition and subtraction, we will consider large or small binary or decimal numbers. For multiplication and division we consider small binary numbers only.**

BINARY ADDITION

First consider multibyte, binary addition.

Two positive, integer numbers, each CNT bytes long, are to be added. The number buffer starting addresses are given by BUF1 and BUF2. The answer is to be stored in a buffer starting at BUF3.

The multibyte addition may be illustrated as follows:

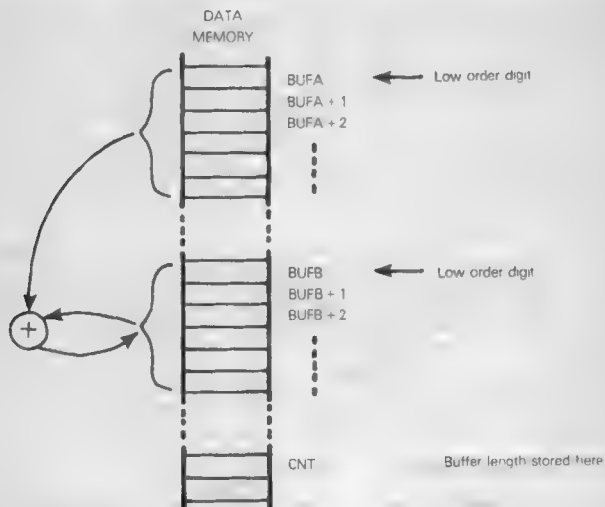


This instruction sequence performs the illustrated addition:

	LDA	CNT	:LOAD BUFFER LENGTH AND SAVE IN C
	MOV	C,A	
	LXI	H,BUFC	:LOAD ANSWER BUFFER ADDRESS INTO H AND L
	PUSH	H	:SAVE ON THE STACK
	LXI	D,BUFA	:LOAD FIRST BUFFER ADDRESS INTO D AND E
	LXI	H,BUFB	:LOAD SECOND BUFFER ADDRESS INTO H AND L
	XRA	A	:CLEAR CARRY
LOOP	LDAX	D	:LOAD NEXT BUF1 BYTE
	ADC	M	:ADD NEXT BUF2 BYTE
	XTHL		:SAVE IN NEXT ANSWER BUFFER BYTE
	MOV	M,A	
	DCX	H	:INCREMENT BUFC ADDRESS
	XTHL		
	DCX	D	:INCREMENT BUF1 ADDRESS

DCX	H	:INCREMENT BUFB ADDRESS
DCR	C	:DECREMENT COUNTER
JNZ	LOOP	:RETURN FOR MORE BYTES

Multibyte addition is simpler if you can store the sum in one of the source buffers:



Here is the shorter instruction sequence:

LDA	CNT	:LOAD BUFFER LENGTH AND SAVE IN C
MOV	C,A	
LXI	D,BUFA	:LOAD FIRST BUFFER ADDRESS INTO D,E
LXI	H,BUFB	:LOAD SECOND AND ANSWER BUFFER ADDRESS INTO H,L
XRA	A	:CLEAR CARRY
LOOP	LDAX	D :LOAD NEXT BUFA BYTE
	ADC	M :ADD NEXT BUFB BYTE
	MOV	M,A :STORE ANSWER
	INX	D :INCREMENT BUFA ADDRESS
	INX	H :INCREMENT BUFB ADDRESS
	DCR	C :DECREMENT BUFFER LENGTH
	JNZ	LOOP :RETURN IF NOT END

BINARY SUBTRACTION

Because the 8080 has special subtraction instructions, binary subtraction is almost identical to binary addition. In either subroutine, simply replace the ADC instruction with the SBB instruction and accurate binary subtraction will result.

DECIMAL ADDITION

Decimal addition is also very easy using an 8080 microcomputer. **Simply insert a DAA instruction to follow the ADC** in either of the binary addition programs and you have decimal addition:

```

      .
      .
      .
LOOP  LDAX    D      ;LOAD NEXT BUF1 BYTE
      ADC     M      ;ADD NEXT BUF2 BYTE
      DAA      ;DECIMAL ADJUST RESULT
      XTHL     ;SAVE IN NEXT ANSWER BUFFER BYTE
      .
      .
      .
```

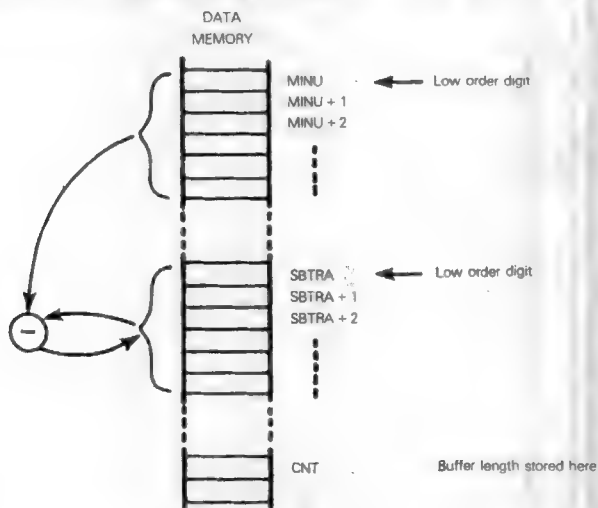
One caution, however: the decimal addition routine you create assumes that valid binary coded decimal data is stored in your source buffers. If, in error, you have invalid data in either of your source buffers, you will generate a meaningless answer — and not know it.

If your program is one which cannot guarantee that data in source buffers is valid binary coded decimal, then you must write a routine to check buffer contents and insure that no high or low 4 bit unit within any byte contains a binary code of A through F.

DECIMAL SUBTRACTION

Decimal subtraction is complicated somewhat by the fact that you cannot use the 8080 subtract instructions; these instructions only work for binary data, since they automatically generate the two's complement of the subtrahend. As described in "An Introduction To Microcomputers", Volume I, binary coded decimal subtraction requires that you take the ten's complement of the subtrahend.

Let us return to the shorter binary addition program and create, in its place, a decimal subtraction equivalent; here is the appropriate memory map:



Here is the required instruction sequence:

```

DSUB: LXI    D,MINU    ;D AND E ADDRESS MINUEND
      LXI    H,SBTRA    ;H AND L ADDRESS SUBTRAHEND
      LDA    CNT    ;LOAD BUFFER LENGTH AND SAVE
      MOV    C,A    ;IN C
      STC    ;SET CARRY INDICATING NO BORROW
LOOP: MVI    A,99H    ;LOAD ACCUMULATOR WITH 99H
      ACI    0    ;ADD ZERO WITH CARRY
      SUB    M    ;PRODUCE NINES COMPLEMENT OF SUBTRAHEND
      XCHG    ;SWITCH D AND E WITH H AND L
      ADD    M    ;ADD MINUEND
      DAA    ;DECIMAL ADJUST ACCUMULATOR
      XCHG    ;RESWITCH D AND E WITH H AND L
      MOV    M,A    ;STORE RESULT
      INX    D    ;ADDRESS NEXT BYTE OF MINUEND
      INX    H    ;ADDRESS NEXT BYTE OF SUBTRAHEND
      DCR    C    ;DECREMENT BYTE COUNT
      JNZ    LOOP    ;GET NEXT 2 DECIMAL DIGITS
DONE: NOP

```

MULTIPLICATION AND DIVISION

Multiplication and division must be approached with an element of caution within microcomputer systems. These are operations which are unsuited to the organization of a microcomputer, any nontrivial multiplication or division can take so long to execute that it will severely degrade overall performance. **If your microcomputer application is going to make extensive use of multiplication, division or transcendental functions, you should seriously consider using one of the many calculator/arithmetic chips that are now commercially available.** Transferring complex arithmetic to such a chip can make the difference between a microcomputer system being viable or nonviable in your application.

You can implement simple multiplication and division in microcomputer systems that do not make extensive, or time-consuming use of these routines; therefore we will describe some simple program sequences.

8-BIT BINARY MULTIPLICATION

Consider the multiplication of two unsigned, 8-bit data values, to generate a 16-bit product. The simplest way of performing this multiplication is to add the multiplier to 0 the number of times given by the multiplicand. For example, **you can multiply 4 by 3 if you add 4 to 0 three times:**

```

      MVI    A,0    ;CLEAR AB TO INITIALIZE ANSWER
      MOV    B,A    ;BUFFER
      CMP    D    ;TEST FOR 0 IN D
      RZ    ;IF 0, ANSWER IS 0 SO END
LOOP: ADD    E    ;ADD MULTIPLIER TO LOW ORDER ANSWER BYTE
      JNC    NEXT    ;IF CARRY IS SET, INCREMENT B
      INR    B
NEXT: DCR    D    ;DECREMENT MULTIPLICAND
      JNZ    LOOP    ;IF NOT ZERO, RETURN TO ADD AGAIN
      RET

```

There is a faster way of executing multiplications. We can use the fact that a binary digit is limited to having values of 0 or 1; this means that at the single digit level, multiplication degenerates to addition or no addition.

Let us explain this concept; using common decimal notation, consider the following multiplication:

$$\begin{array}{r}
 142 \quad \text{Multiplicand} \\
 \times 307 \quad \text{Multiplier} \\
 \hline
 42600 \\
 0000 \quad \text{Partial Product} \\
 994 \quad \text{Partial Product} \\
 \hline
 43594 \quad \text{Product}
 \end{array}$$

142 = Multiplier

307 = Multiplicand

Add 7 x Multiplier to product
 Shift Multiplier two digits left, then
 multiply by 3 and add to product

Each partial product equals the multiplicand being multiplied by one digit of the multiplier. The partial product is shifted to the left by tacking on 0's to the right. The number of 0's tacked on to the right is equal to the number of digits to the right of the current multiplier digit:

$$\begin{array}{r}
 142 \\
 3 \times x \\
 \hline
 42600
 \end{array}$$

Two 0s tacked on since there are two digits to the right of 3

142 x 3

We can extend this same concept to binary arithmetic, in which case the problem becomes very simple, since no binary digit can have a value other than 0 or 1. This being the case, you have only two choices: wherever a multiplier digit is 0, you do not add the shifted multiplicand to the answer; but if the multiplier digit is 1 you do add the shifted multiplicand to the answer. Here is an example:

$$\begin{array}{r}
 10110101 = \text{Multiplicand (M)} \\
 01101101 = \text{Multiplier}
 \end{array}$$

Add M to product
 Shift M two digits left and add
 Shift M three digits left and add
 Shift M five digits left and add
 Shift M six digits left and add

Product =

$$\begin{array}{r}
 10110101 \\
 1011010100 \\
 1110001001 \\
 10110101000 \\
 100100110001 \\
 1011010100000 \\
 11111111010001 \\
 10110101000000 \\
 \hline
 100110100010001 \\
 \hline
 4 \quad D \quad 4 \quad 1 \quad 1 \quad 1
 \end{array}$$

$10110101 \times 01101101 = 0100110100010001$
 B 5 6 D 4 D 1 1

Using the "shift-and-add" technique, the following steps will multiply a one-byte multiplicand by a one-byte multiplier to produce the correct two-byte result:

- Test the least significant bit of the multiplier. If zero, go to Step b. If one, add the multiplicand to the most significant byte of the result.
- Shift the entire two-byte result right one bit position.
- Repeat Steps a and b until all 8 bits of the multiplier have been tested.

Consider $B5 \times 6D$, the binary multiplication we just illustrated:

Multiplier = 01101101
 Multiplicand = 10110101

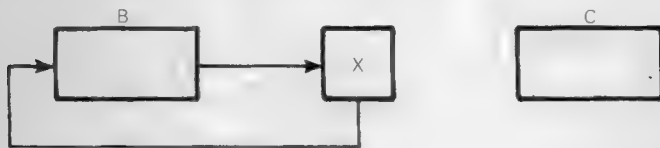
		RESULT	
		HIGH ORDER BYTE	LOW ORDER BYTE
	Start	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
	Step 1 (a)	1 0 1 1 0 1 0 1	0 0 0 0 0 0 0 0
	1 (b)	0 1 0 1 1 0 1 0	1 0 0 0 0 0 0 0
	Step 2 (a,b)	0 0 1 0 1 1 0 1	0 1 0 0 0 0 0 1
	Step 3 (a)	1 0 1 1 0 1 0 1	
	3 (b)	0 1 1 1 0 0 0 1	0 0 1 0 0 0 0 0
	Step 4 (a)	1 0 1 1 0 1 0 1	
	4 (b)	0 0 1 0 0 1 1 0	0 0 1 0 0 0 0 0
	Step 5 (a,b)	0 1 0 0 1 0 0 1	1 0 0 0 1 0 0 0
	Step 6 (a)	1 0 1 1 0 1 0 1	
	6 (b)	0 1 1 1 1 1 1 0	0 1 0 0 0 1 0 0
	Step 7 (a)	1 0 1 1 0 1 0 1	
	7 (b)	0 1 1 1 0 1 0 0	0 0 1 1 1 0 1 0
	Step 8 (a)	0 1 0 0 1 1 0 1	0 0 0 1 1 0 1 0

We will now write a program to implement this multiplication algorithm.

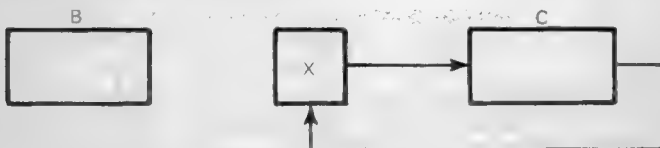
The B register will hold the most significant byte of the result, and the C register will hold the least significant byte of the result.

The 16-bit right shift of the result is performed by two rotate right-through-carry instructions as follows:

Rotate B:



Then rotate C to complete the shift



Register D holds the multiplicand, and register C originally holds the multiplier. Here is the program:

```

MULT    MVI    B,0           ;INITIALIZE MOST SIGNIFICANT BYTE
                                ;OF RESULT
                                ;BIT COUNTER
MULT0   MVI    E,9           ;BIT COUNTER
        MOV    A,C           ;ROTATE LEAST SIGNIFICANT BIT OF
        RAR                    ;MULTIPLIER TO CARRY AND SHIFT
        MOV    C,A           ;LOW ORDER BYTE OF RESULT
        DCR    E
        JZ     DONF          ;EXIT IF COMPLETE
        MOV    A,B
        JNC    MULT1
        ADD    D              ;ADD MULTIPLICAND TO HIGH ORDER BYTE
                                ;OF RESULT IF BIT WAS A ONE
MULT1   RAR                    ;CARRY=0 HERE; SHIFT HIGH ORDER
                                ;BYTE OF RESULT
        MOV    B,A
        JMP    MULT0

```

DONE

8-BIT BINARY DIVISION

An analogous procedure is used to divide an unsigned 16-bit number by an unsigned 8-bit number. Here, **the process involves subtraction rather than addition, and rotate-left instructions instead of rotate-right instructions.**

The program uses the B and C registers to hold the most and least significant byte of the dividend respectively; the D register holds the divisor. The 8-bit quotient is generated in the C register, and the remainder is generated in the B register.

```

DIV     MVI    E,9           ;BIT COUNTER
        MOV    A,B
DIV0    MOV    B,A
        MOV    A,C           ;ROTATE CARRY INTO C REGISTER, ROTATE
        RAL                    ;NEXT MOST SIGNIFICANT BIT TO CARRY
        MOV    C,A
        DCR    F
        JZ     DIV1
        MOV    A,B           ;ROTATE MOST SIGNIFICANT BIT TO
        RAL                    ;HIGH ORDER QUOTIENT
        SUB    D              ;SUBTRACT DIVISOR. IF LESS THAN
        JNC    DIV0           ;HIGH ORDER QUOTIENT, GO TO DIV0
        ADD    D              ;OTHERWISE ADD IT BACK
        JMP    DIV0
DIV1    RAL
        MOV    E,A
        MVI    A,FFH,        ;COMPLEMENT THE QUOTIENT
        XRA    C
        MOV    C,A
        MOV    A,E
        RAR

```

DONE

16-BIT BINARY MULTIPLICATION

Now consider the multiplication of two 16-bit numbers, yielding a 32-bit result.

This is the algorithms used:

- 1) Shift the multiplier (HO 16 bits) and partial product (LO 16 bits) left through the carry
- 2) Add the 16-bit multiplicand to three bytes of the partial product if a one bit was shifted out of the multiplier into the carry

Here are the necessary instructions:

```

MPY    LXI    H,0      :INITIALIZE PARTIAL PRODUCT IN HL TO ZERO
        MVI    A,16     :INITIALIZE COUNT
LOOP   DAD    H        :ADD HL TO HL -- LEFT SHIFT LOGICAL INTO CARRY
        XCHG   :EXCHANGE HL AND DE
        JC     MPY1     :JUMP IF CARRY OUT FROM HL
        DAD    H        :NO CARRY -- SHIFT MULTIPLIER LEFT LOGICAL INTO CARRY
        JMP    MPY2     :JUMP
MPY1   DAD    H        :CARRY-SHIFT MULTIPLIER LEFT LOGICAL INTO CARRY
        INX    H        :AND INCREMENT
MPY2   XCHG   :REPOINT TO PARTIAL PRODUCT
        JNC    MPY3     :JUMP IF NO ADD (MULTIPLIER BIT IN CARRY=0)
        DAD    B        :ADD MULTIPLICAND IN BC TO PARTIAL PRODUCT IN HL
        JNC    MPY3     :JUMP IF NO CARRY OUT
        INX    D        :INCREMENT DE TO ADD CARRY
MPY3   DCR    A        :DECREMENT COUNT
        JNZ    LOOP     :LOOP BACK IF NOT ZERO
        RET          :RETURN
    
```

BINARY DIVISION

Consider simple 8-bit division. $B3_{16}$ divided by 15_{16} may be illustrated as follows:

$$\begin{array}{r}
 \text{Divisor} \longrightarrow 10101 \overline{) 10110011} \\
 \underline{10101} \\
 1011
 \end{array}$$

1000 ← Quotient
10110011 ← Dividend
1011

The result is 8_{16} with a remainder of B_{16} .

The division algorithm works by shifting the dividend left into a register that is initially cleared. Whenever the dividend shift buffer contents exceeds the divisor, the divisor is subtracted from the shift buffer contents and a binary 1 digit is inserted into the appropriate quotient bit position.

Consider the following register assignments:



Initially assume that the divisor is in Register E and the dividend is in Register C. The quotient will be generated in Register L. Here is the division program which results:

```

INITIALLY CLEAR REGISTERS A, B, L AND CARRY
    XRA    A           ;EXCLUSIVE OR A WITH ITSELF. THIS CLEARS A
    MOV    B,A         ;CLEAR B
    MOV    L,A         ;CLEAR L
INITIALIZE BIT COUNTER IN REGISTER C
    MVI    C,80H
;SHIFT B AND C, AS A 16-BIT UNIT, ONE BIT LEFT
LOOP    MOV    A,C
        RAL
        MOV    C,A
        MOV    A,B
        RAL
        MOV    B,A
;COMPARE DIVISOR (IN E) WITH DIVIDEND, SHIFT BUFFER,
;CURRENTLY STILL IN A
        CMA    E
        JC     NEXT    ;IF DIVISOR IS LARGER, BYPASS SUBTRACT
;DIVISOR IS SMALLER, SUBTRACT FROM
;DIVIDEND SHIFT BUFFER
        SUB    E
        MOV    B,A
;SET TO 1 CURRENT BIT OF L. THE CURRENT BIT IS
;THE 1 BIT POSITION IN D
        MOV    A,D
        ORA    L
        MOV    L,A
;SHIFT D RIGHT ONE BIT POSITION AND CLEAR CARRY
NEXT    MOV    A,D
        RRC
        JNC    LOOP    ;IF CARRY IS NOT SET, RETURN FOR NEXT BIT

```

At the end, the quotient will be in L, while the remainder is in B

PROGRAM EXECUTION SEQUENCE LOGIC

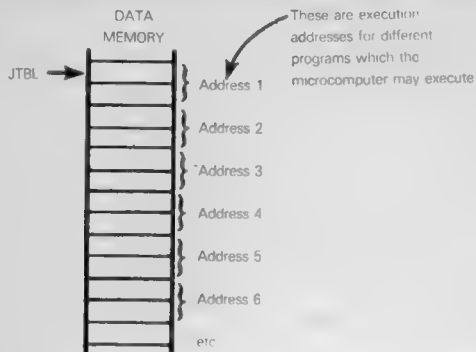
THE JUMP TABLE

There is really only one program sequence that needs to be described under this heading: it is the Jump Table.

Remember that the 8080 instruction set is rich in conditional instructions; Jump, Call and Return instructions all have eight conditional variations, which means that special routines are not required when your logic can go one of two ways only.

When you have three or more options, the Jump Table becomes an effective programming tool.

At the heart of a Jump Table there will be a sequence of 16-bit addresses stored in pairs of contiguous memory bytes:



We will presume that these contiguous memory addresses represent the starting addresses for a number of different programs. Assuming that the required program is identified by a program number in the Accumulator, **the following instruction sequence causes execution to branch to the program whose number is stored in the Accumulator:**

JUMP TABLE PROGRAM

LXI	H, JTBL	:LOAD JUMP TABLE BASE ADDRESS IN HL
RLC		:MULTIPLY ACCUMULATOR BY 2
ADD	L	:ADD LOW ORDER ADDRESS BYTE TO A
MOV	L, A	:RESTORE SUM TO L
MVI	A, 0	:ADD CARRY (IF ANY) TO H
ADC	H	
MOV	H, A	
MOV	E, M	:HL ADDRESSES REQUIRED ADDRESS
INX	H	:LOAD REQUIRED ADDRESS INTO D,E
MOV	D, M	
XCHG		:MOVE ADDRESS FROM DE TO HL
PCHL		:MOVE ADDRESS FROM HL TO PC

APPENDIX A

STANDARD CHARACTER CODES

Hexadecimal Representation	ASCII (7 bit)	EBCDIC (8 bit)	Hexadecimal Representation	ASCII (7bit)	EBCDIC (8 bit)
0			31	!	
1			32	"	
2			33	#	
3			34	\$	
4			35	%	
5			36	&	
6			37	'	
7			38	(
8			39)	
9			3A	*	
A			3B	+	
B			3C	,	
C			3D	-	
D			3E	.	
E			3F	/	
F			40	@	blank
10			41	A	
11			42	B	
12			43	C	
13			44	D	
14			45	E	
15			46	F	
16			47	G	
17			48	H	
18			49	I	
19			4A	J	
1A			4B	K	
1B			4C	L	
1C			4D	M	
1D			4E	N	
1E			4F	O	
1F			50	P	
20	blank		51	Q	
21	!		52	R	
22	"		53	S	
23	#		54	T	
24	\$		55	U	
25	%		56	V	
26	&		57	W	
27	'		58	X	
28	(59	Y	
29)		5A	Z	
2A	*		5B	[
2B	+		5C	\	
2C	,		5D]	
2D	-		5E	^	
2E	.		5F	_	
2F	/		60	a	
30	0		61		

APPENDIX A (continued)

Hexadecimal Representation	ASCII (7 bit)	EBCDIC (8 bit)	Hexadecimal Representation	ASCII (7bit)	EBCDIC (8 bit)
62	b		97		p
63	c		98		q
64	d		99		r
65	e		9A		
66	f		9B		
67	g		9C		
68	h		9D		
69	i		9E		
6A	j		9F		
6B	k	,	A0		
6C	l	%	A1		
6D	m	-	A2		s
6E	n)	A3		t
6F	o	?	A4		u
70	p		A5		v
71	q		A6		w
72	r		A7		x
73	s		A8		y
74	t		A9		z
75	u		AA		
76	v		AB		
77	w		AC		
78	x		AD		
79	y		AE		
7A	z		AF		
7B		#	B0		
7C		@	B1		
7D		.	B2		
7E		=	B3		
7F		^	B4		
80			B5		
81		a	B6		
82		b	B7		
83		c	B8		
84		d	B9		
85		e	BA		
86		f	BB		
87		g	BC		
88		h	BD		
89		i	BE		
8A			BF		
8B			C0		
8C			C1		A
8D			C2		B
8E			C3		C
8F			C4		D
90			C5		E
91			C6		F
92		j	C7		G
93		k	C8		H
94		l	C9		I
95		m	CA		
96		n	CB		
		o			

APPENDIX A (continued)

Hexadecimal Representation	ASCII (7 bit)	EBCDIC (8 bit)	Hexadecimal Representation	ASCII (7bit)	EBCDIC (8 bit)
CC			E6		W
CD			E7		X
CE			E8		Y
CF			E9		Z
D0			EA		
D1		J	EB		
D2		K	EC		
D3		L	ED		
D4		M	EE		
D5		N	EF		
D6		O	F0		0
D7		P	F1		1
D8		Q	F2		2
D9		R	F3		3
DA			F4		4
DB			F5		5
DC			F6		6
DD			F7		7
DE			F8		8
DF			F9		9
E0			FA		
E1			FB		
E2		S	FC		
E3		T	FD		
E4		U	FE		
E5		V	FF		

ABOUT THE AUTHORS OF THIS BOOK

Adam Osborne is president of Osborne and Associates, Inc., a California corporation.

Osborne and Associates, Inc., are microcomputer consultants. We will design your microcomputer based product for you, or we will help you do the job for yourself. We also deliver custom, in-house seminars on microcomputers, their future potential or their immediate use.

For manufacturers in the microcomputer and minicomputer industries, Osborne and Associates prepare technical manuals.

To order additional copies of this book, or to inquire about our services, write or telephone:

Osborne and Associates, Inc.
P.O. Box 2036
Berkeley, California 94702
(415) 548-2805

OTHER BOOKS IN THIS SERIES

- | | |
|------|--|
| 2001 | AN INTRODUCTION TO MICROCOMPUTERS
VOLUME 1: BASIC CONCEPTS
\$7.50 AVAILABLE: May 31, 1976 |
| 3001 | AN INTRODUCTION TO MICROCOMPUTERS
VOLUME 2: SOME REAL PRODUCTS
\$7.50 AVAILABLE: July 15, 1976 |
| 5001 | 6800 PROGRAMMING FOR LOGIC DESIGN
\$7.50 AVAILABLE: October 1, 1976 |

\$7.50